LINUX DEVICE DRIVERS

第三版

LINUX 设备驱动程序



JONATHAN CORBET, ALESSANDRO RUBINI
& GREG KROAH-HARTMAN 著

魏永明 耿岳 钟书毅 译

O'REILLY®
中国电力出版社

WHERE THE KERNEL MEETS THE HARDWARE

LINUX设备驱动程序



本书是经典著作《Linux设备驱动程序》的第三版。如果您希望在Linux操作系统上支持计算机外部设备，或者在Linux上运行新的硬件，或者只是希望一般性地了解Linux内核的编程，就一定要阅读本书。本书描述了如何针对各种设备编写驱动程序，而在过去，这些内容仅仅以口头形式交流，或者零星出现在神秘的代码注释中。

本书的作者均是Linux社区的领导者。**Jonathan Corbet**虽不是专职的内核代码贡献者，但他是备受关注的LWN.net新闻及信息网站的执行编辑。**Alessandro Rubini**是一名Linux代码贡献者，也是活跃的意大利Linux社区的灵魂人物。**Greg Kroah-Hartman**是目前内核中USB、PCI和驱动程序核心子系统（本书均有讲述）的维护者。

本书的这个版本已针对Linux内核的2.6.10版本彻底更新过了。内核的这个版本针对常见任务完成了合理化设计及相应的简化，如即插即用、利用sysfs文件系统和用户空间交互，以及标准总线上的多设备管理等等。

要阅读并理解本书，您不必首先成为内核黑客；只要您理解C语言并具有Unix系统调用的一些背景知识即可。您将学到如何为字符设备、块设备和网络接口编写驱动程序。为此，本书提供了完整的示例程序，您不需要特殊的硬件即可编译和运行这些示例程序。本书还在单独的章节中讲述了PCI、USB和tty（终端）子系统。对期望了解操作系统内部工作原理的读者来讲，本书也深入阐述了地址空间、异步事件以及I/O等方面的内容。

本书涵盖的主题包括：

- 完整的字符、块、tty（终端）及网络驱动程序
- 驱动程序的调试
- 中断
- 计时问题
- 并发、锁定和对称多处理器系统（SMP）
- 内存管理和DMA
- 驱动程序模型和sysfs
- 热插拔设备
- 对常见总线的描述，包括SCSI、PCI、USB和IEEE1394（火线）

ISBN 7-5083-3863-4

9 787508 338637 >

O'REILLY®

www.oreilly.com.cn

O'Reilly Media, Inc.授权中国电力出版社出版

ISBN 7-5083-3863-4

定价：69.00元

LINUX
设备驱动程序

第三版

Jonathan Corbet,
Alessandro Rubini &
Greg Kroah-Hartman 著
魏永明 耿岳 钟书毅 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc. 授权中国电力出版社出版

中国电力出版社

图书在版编目（CIP）数据

Linux设备驱动程序：第3版/（美）科波特（Corbet, J.）等著；魏永明，耿岳，钟书毅译．－北京：中国电力出版社，2005.11

书名原文：Linux Device Drivers, Third Edition

ISBN 7-5083-3863-4

I. L... II. ①科 ... ②魏 ... ③耿 ... ④钟 ... III. Linux操作系统 IV. TP316.89

中国版本图书馆CIP数据核字（2005）第116363号

北京市版权局著作权合同登记

图字：01-2005-5621号

©2005 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Electric Power Press, 2005. Authorized translation of the English edition, 2005 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由O'Reilly Media, Inc. 出版2005。

简体中文版由中国电力出版社出版2005。英文原版的翻译得到O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

书　　名/ Linux设备驱动程序（第三版）

书　　号/ ISBN 7-5083-3863-4

责任编辑/ 陈维宁 牛贵华

封面设计/ Edie Freedman，张健

出版发行/ 中国电力出版社（www.infopower.com.cn）

地　　址/ 北京三里河路6号（邮政编码100044）

经　　销/ 全国新华书店

印　　刷/ 北京市地矿印刷厂

开　　本/ 787毫米×980毫米　16开本　36.75印张　597千字

版　　次/ 2006年1月第一版　2006年1月第一次印刷

印　　数/ 0001-4000册

定　　价/ 69.00元（册）

O'Reilly Media, Inc. 介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构O'Reilly Media, Inc. 授权中国电力出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在 UNIX、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时是联机出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》（被纽约公共图书馆评为二十世纪最重要的50本书之一）到GNN（最早的Internet门户和商业网站），再到WebSite（第一个桌面PC的Web服务器软件），O'Reilly Media, Inc.一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc. 是最稳定的计算机图书出版商 —— 每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得 O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc. 还有许多固定的作者群体 —— 他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc. 依靠他们及时地推出图书。因为O'Reilly Media, Inc. 紧密地与计算机业界联系着，所以O'Reilly Media, Inc. 知道市场上真正需要什么图书。

目录

前言 1

第一章 设备驱动程序简介 9

设备驱动程序的作用 10
内核功能划分 12
设备和模块的分类 14
安全问题 15
版本编号 17
许可证条款 18
加入内核开发社团 19
本书概要 19

第二章 构造和运行模块 21

设置测试系统 21
Hello World 模块 22
核心模块与应用程序的对比 24
编译和装载 28
内核符号表 33
预备知识 35
初始化和关闭 36

模块参数 40
在用户空间编写驱动程序 42
快速参考 44

第三章 字符设备驱动程序 46
scull 的设计 46
主设备号和次设备号 47
一些重要的数据结构 53
字符设备的注册 59
open 和 release 62
scull 的内存使用 64
read 和 write 67
试试新设备 74
快速参考 74

第四章 调试技术 76
内核中的调试支持 76
通过打印调试 78
通过查询调试 85
通过监视调试 94
调试系统故障 96
调试器和相关工具 102

第五章 并发和竞态 109
scull 的缺陷 109
并发及其管理 110
信号量和互斥体 111
completion 116
自旋锁 118
锁陷阱 123

除了锁之外的办法 125
快速参考 132

第六章 高级字符驱动程序操作 137

ioctl 137
阻塞型 I/O 149
poll 和 select 163
异步通知 168
定位设备 172
设备文件的访问控制 173
快速参考 179

第七章 时间、延迟及延缓操作 183

度量时间差 183
获取当前时间 188
延迟执行 190
内核定时器 196
tasklet 202
工作队列 204
快速参考 208

第八章 分配内存 213

kmalloc 函数的内幕 213
后备高速缓存 217
get_free_page 和相关函数 221
vmalloc 及其辅助函数 225
per-CPU 变量 228
获取大的缓冲区 230
快速参考 231

第九章 与硬件通信 235

I/O 端口和 I/O 内存 235
使用 I/O 端口 239
I/O 端口示例 245
使用 I/O 内存 248
快速参考 254

第十章 中断处理 258

准备并口 259
安装中断处理例程 259
实现中断处理例程 269
顶半部和底半部 274
中断共享 278
中断驱动的 I/O 281
快速参考 285

第十一章 内核的数据类型 287

使用标准 C 语言类型 287
为数据项分配确定的空间大小 289
接口特定的类型 289
其他有关移植性的问题 291
链表 294
快速参考 298

第十二章 PCI 驱动程序 300

PCI 接口 300
ISA 回顾 317
PC/104 和 PC/104+ 319
其他的 PC 总线 319
SBus 320

NuBus 321
外部总线 321
快速参考 322

第十三章 USB 驱动程序 324
USB 设备基础 326
USB 和 Sysfs 329
USB urb 331
编写 USB 驱动程序 342
不使用 urb 的 USB 传输 352
快速参考 356

第十四章 Linux 设备模型 359
kobject、kset 和子系统 361
低层 sysfs 操作 368
热插拔事件的产生 372
总线、设备和驱动程序 374
类 384
各环节的整合 388
热插拔 394
处理固件 401
快速索引 403

第十五章 内存映射和 DMA 408
Linux 的内存管理 408
mmap 设备操作 418
执行直接 I/O 访问 429
直接内存访问 435
快速参考 453

第十六章 块设备驱动程序 458

注册 459
块设备操作 464
请求处理 468
其他一些细节 484
快速参考 487

第十七章 网络驱动程序 491

snull 设计 492
连接到内核 495
net_device 结构细节 499
打开和关闭 508
数据包传输 510
数据包的接收 514
中断处理例程 516
不使用接收中断 518
链路状态的改变 521
套接字缓冲区 521
MAC 地址解析 525
定制 ioctl 命令 527
统计信息 528
组播 529
其他知识点详解 533
快速参考 534

第十八章 TTY 驱动程序 538

小型 TTY 驱动程序 540
tty_driver 函数指针 545
TTY 线路设置 551
ioctls 555

proc 和 sysfs 对 TTY 设备的处理 557
tty_driver 结构详解 558
tty_operations 结构详解 560
tty_struct 结构详解 562
快速参考 564

参考书目 567



前言

顾名思义，本书是讲述如何编写Linux设备驱动程序的。面对层出不穷的新硬件产品，必须有人不断编写新的驱动程序以便让这些设备能够在Linux下正常工作，从这个意义上讲，讲述驱动程序的编写本身就是一件非常有意义的工作。但本书也涉及到Linux内核的工作原理，同时还讲述如何根据自己的需要和兴趣来定制Linux内核。Linux是一个开放的系统，我们希望借助本书使它能够更加开放，从而能够吸引更多的开发人员。

本书是《Linux设备驱动程序》的第三版。自本书第一版发行以来，内核已经发生了巨大变化，我们必须努力让本书跟上内核的发展步伐。在这一版本中，我们尽可能完整地描述了2.6.10内核。这次，我们决定略去针对先前内核版本的向后兼容性描述，这是因为从2.4以来内核发生的改变实在太大了，而针对2.4内核的接口描述在本书第二版（可免费获得）中有很好的阐述。

这一版本包括了一些2.6内核相关的新内容。关于锁和并发性的内容得到了进一步充实，而且单独成章。我们还详细描述了2.6内核中新引入的Linux设备模型。我们用新的章节来描述USB总线和串行驱动程序子系统；同时，讲述PCI的那一章也得到了加强。本书其余部分类似先前的版本，但几乎每一章都彻底更新过了。

我们希望读者能够从本书的学习中获得乐趣，就像我们自己从编写本书的过程中获得乐趣一样。

Jon的介绍

这个版本出版的时候，恰好我在Linux界工作了12年，更惊人的是恰好是我在计算机领域工作的第25个年头。1980年时，计算机领域就已经是个快速发展的领域，然而此后又加速不少。让本书保持更新状态面临越来越大的挑战；Linux内核黑客在不停地增强他们的代码，但很少有耐心去关心文档是否跟得上步伐。

在市场上Linux保持着成功，但更重要的是Linux赢得了全球开发人员的关注。很明显，Linux的成功证明了其优秀的技术质量以及自由软件的大量好处。但在我看来，其成功的真正关键在于如下事实：Linux将快乐重新带回到计算机领域。利用Linux，任何人可以了解系统并以任何可能的方式贡献自己的代码，当然，代码在技术上的优势是其中最有价值的。Linux不仅为我们提供了一个顶级质量的操作系统，而且也为我们提供了参与到其未来开发过程的机会，我们完全可以从中得到无尽的快乐。

在计算机领域的25年中，我曾经有过许多有意思的经历，从第一次为Cray计算机编程（用Fortran语言在纸带上打孔），到亲历迷你计算机和Unix工作站的变迁，一直到当前微处理器占支配地位的时代。我还没有看到哪个领域可以让人如此着迷并因此开心快乐，也从未有过像现在这样能够完全控制我们的工具及其发展的时候。很明显，Linux和自由软件是这些变化背后的驱动力。

我希望本书能将这种快乐和机会带给新的Linux开发人员。不管你的兴趣在内核级还是用户空间，我都希望本书是一本有用而且有趣的指南，它能够帮助读者发现内核是如何和硬件一同工作的。我希望本书能帮助和启发读者利用自己的编辑器让我们共享的、自由的操作系统更加美好！Linux已经走过了很长的路，然而此刻也正是起点，观察并参与其中将为你带来更大的乐趣。

Alessandro的介绍

我一直喜欢玩电脑，就因为通过电脑我可以控制外部的硬件。我曾为Apple II和ZX Spectrum系统焊接我自己的设备，之后，有了大学中学到的Unix和自由软件专业知识，通过在新的386系统上安装了GNU/Linux并再次玩起了自己的电路板，我逃离了DOS陷阱。

那时Linux社区还非常小，也没有太多的文档来描述如何编写驱动程序，于是我开始为《Linux Journal》撰稿。这就是事情的开端：当我发现我自己不喜欢撰写论文后，我离开了大学，并和O'Reilly签订了本书第一版的编写合同。

这是1996年的事情，已经过去好多年了。

现在，计算机世界已经大不相同了：自由软件已经成为一种可行方案，不论在技术上还是在政治上，然而在这两个领域仍然有许多工作要做。我希望本书能够促进如下两个目标的实现：传播技术知识并提高对传播知识必要性的认同。这也是本书第一版被大众广泛接受以来第二版的两位作者在编辑和出版商的支持下转向自由许可证的原因。我坚信这是正确的知识传播途径，并且有利于和认同这种观点的其他人合作。

在嵌入式领域中所发生的一切令我兴奋，我希望本书能够为Linux的应用推波助澜；然而，在今天这个时代，思想的变化尤其迅速，为第四版作计划的时间已经到来，我们也正在寻求第四位作者的帮助。

Greg的介绍

从为了编写一个真实的Linux驱动程序而拿起《Linux设备驱动程序》第一版到现在，已经过很长一段时间。本书第一版帮助我理解了Linux操作系统的内部细节，而在此之前，我使用该操作系统有很长的时间，但几乎没有时间来研究内核细节。有了第一版中获得的知识，加上阅读内核中其他程序员的代码，我的第一个充满缺陷、非SMP安全的驱动程序被内核社区接受，并加入到了内核代码的主分支中。尽管在五分钟之后我就收到了我的第一个缺陷报告，但自此我就被希望尽我所能使Linux操作系统成为最好的欲望吸引住了。

我非常荣幸能够为本书贡献一些东西。我希望本书能够帮助其他人掌握与内核相关的一些细节，你会发现驱动程序的开发并不像想像的那么可怕或吓人，当然，我也希望本书能够鼓励其他人加入或者帮助这个大的集体，以便让这个操作系统可以在每一个计算机平台上运行，并支持每一种可获得的设备。开发过程充满着乐趣，加入社区非常值得，因为每个人都能从努力付出中获得好处。

现在，让我们一起寻找这个版本的缺陷，修改API以便让它们更好地工作，或者更加简单而便于每个人理解，或者增加新的特性。来吧，参与其中我们也将得到他人的帮助。

本书的读者对象

本书对那些希望编写计算机设备驱动程序的人员、或者那些要解决Linux机器内部问题的程序员来讲，将是非常有帮助的。请注意，“Linux机器”是一个比“运行Linux的PC”更为宽泛的概念，因为Linux现在能够支持许多不同的硬件平台，而内核编程不再绑定到某个特定的平台。我们希望本书能够成为那些想成为内核黑客但却不知如何下手的人们的良好起点。

在技术方面，本书为理解内核内幕以及理解一些Linux开发者所做出的设计决策支了一招。尽管本书的主要目的是告诉读者如何编写设备驱动程序，但同时也给出了内核实现方面的概览。

尽管真正的黑客能够从正式的内核源代码中找到所有必要的信息，但通常来讲，编写好

的书籍能够更好地帮助读者提高编程技巧。读者将要看到的文字来自对内核源代码的仔细分析，我们希望我们所付出的努力是值得的。

Linux 发烧友可从本书找到深入内核代码的足够精神食粮；通过本书的学习，将有能力加入到为某个新功能或性能增强不停工作的开发小组当中。本书并没有涵盖 Linux 内核的全部，但是作为 Linux 设备驱动程序开发人员，你需要的是了解如何与许多的内核子系统一起工作。因此，本书对内核编程作了一个一般性的介绍。Linux 仍然在不断改进和发展，因此新程序员始终有机会加入到这一 Linux 的开发大军中。

另一方面，如果你只是为了给自己的设备编写一个驱动程序，而不想过多了解内核的内幕信息，本书内容则足够模块化以满足你的需求。如果你不想深入到细节当中，则可以跳过大部分的技术章节，而直接阅读可由设备驱动程序使用的、能够和内核的其他部分无缝结合的标准 API。

内容的组织

本书内容由简到难，并划分为两大部分。第一部分（第一章到第十一章）首先讲述了如何编写内核模块，然后讲述了编写功能完备的字符设备驱动程序所涉及的各个编程主题。每一章讲述一个特定问题，并在每章结尾包含一个“快速小结”，该“快速小结”可在实际开发中作为参考使用。

在本书第一部分中，内容从面向软件的概念过渡到硬件相关的概念。这种组织方法意味着你能够尽可能不在机器中插入任何外部硬件而测试示例代码。每章都包含有源代码，并给出了能够在任意一台 Linux 计算机上运行的示例驱动程序。但是，在第十章和第十一章中，我们需要读者在并口上连接一些电线，以便测试硬件处理代码，当然，这一要求对任何人来讲都是可以做到的。

本书的第二部分（第十二章到第十八章）讲述了块设备驱动程序和网络接口，并深入讨论了一些更加高级的内容，比如虚拟内存子系统和 PCI、USB 总线等。许多驱动程序作者可能不需要这些内容，但我们鼓励你阅读这些章节。尽管对某个特定的项目来说你并不需要了解这些知识，但第二部分的许多内容和了解 Linux 内核的工作原理一样重要。

背景信息

为了更好地利用本书，我们希望读者熟悉 C 语言编程。因为我们经常会提到 Unix 系统调用、命令和管道，因此也需要读者拥有 Unix 的使用经验。

在硬件级，不需要读者有任何预先的经验就可以理解本书内容，当然，一些一般性的概念是必须清楚的。本书内容并不基于某个特定的PC硬件，我们在提到某个特定的硬件时会提供给读者所有必要的信息。

构造内核需要一些自由软件工具，而且经常要求使用这些工具的特定版本。太老的工具可能缺少一些必要的特性，而太新的工具又可能会偶尔生成不能正常工作的内核。通常而言，当前流行的Linux发行版所提供的工具能够很好地工作。不同的内核版本对工具的版本需求不同，这时，你可以参考内核源代码树中的*Documentation/Changes*文件。

在线版本和条款

本书作者已经选择本书在Creative Commons “Attribution-ShareAlike”许可证版本2.0的保护下免费获得:

http://www.oreilly.com/catalog/linuxdrive3

排版约定

下面是本书中用到的一些排版约定:

斜体（*Italic*）
: 用于文件和目录的名称、程序和命令的名称、命令行选项、URL以及新的术语

等宽字体（Constant Width）
: 用于在示例中显示文件内容或者命令的输出，还用于正文中出现的C代码或者其他字符串

等宽斜体（*Constant Width Italic*）
: 用于可变选项、关键词或者需要用户用实际值替换的文字

等宽黑体（**Constant Width Bold**）
: 用于示例中需要用户照原文键入的命令或者其他文字

代码示例的使用

本书用于帮助读者完成自己的任务。通常来讲，读者可以在自己的程序和文档中使用本书中的代码。示例代码采用BSD/GPL双许可证发布。

我们希望（但并不要求）你在代码中增加归属信息。归属信息通常包含标题、作者、出版商以及ISBN。比如："Linux Device Drivers, Third Edition, by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. Copyright 2005 O'Reilly Media, Inc., 0-596-00590-3."

意见和建议

请将有关本书的评论和问题发送给出版商，联系方法如下：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

100080 北京市海淀区知春路 49 号希格玛公寓 B 座 809 室
奥莱理软件（北京）有限公司

我们还为本书建立了一个网页，其中列出了勘误、示例等内容。该网页地址如下：

http://www.oreilly.com/catalog/linuxdrive3

如果你希望对本书进行评论，或者遇到有关本书的技术问题，可发电子邮件到：

info@mail.oreilly.com.cn
bookquestions@oreilly.com

有关 O'Reilly 的更多信息，包括图书、会议、资源中心以及 O'Reilly Network，可访问我们的 Web 站点：

http://www.oreilly.com
http://www.oreilly.com.cn

Safari Enabled



如果你在自己喜欢的技术书籍封面上看到 Safari® Enabled 标识，则说明本书可通过 O'Reilly Network Safari Bookshelf 在线获得。

Safari 提供了比电子书更好的一种方案。它是一种虚拟图书馆，可让读者轻松搜索大量的顶级技术书籍、剪切和粘贴代码示例、下载图书章节并在需要最精确和最新的信息时快速找到答案。请点击 *http://safari.oreilly.com* 免费尝试。

致谢

本书的编写得到了许多人的帮助，我们向他们致以诚挚的谢意，是他们的帮助使本书得以出版。

首先感谢我们的编辑Andy Oram，通过他的努力本书才变成更好的产品。当然我们还要感谢那些建立当前自由软件时代的哲学和实践基础的人物。

本书第一版由Alan Cox、Greg Hankins、Hans Lermen、Heiko Eissfeldt以及Miguel de Icaza（依照名字字母排序）进行了技术审校。第二版的技术审校是Allan B. Cruse、Christian Morgner、Jake Edge、Jeff Garzik、Jens Axboe、Jerry Cooperstein、Jerome Peter Lynch、Michael Kerrisk、Paul Kinzelman和Raph Levien。第三版的技术审校是Allan B. Cruse、Christian Morgner、James Bottomley、Jerry Cooperstein、Patrick Mochel、Paul Kinzelman以及Robert Love。他们花费了大量精力寻找本书的错误或者问题，并且指出了文中可以提高的地方。

最后，让我们感谢Linux开发人员所做出的艰苦工作。这包括内核程序员以及经常会被遗忘的应用软件开发人员。本书中，我们选择不提到他们的名字，以避免因为遗忘其他人的名字而显得不公平。当然也有例外，我们会提到Linus的名字，希望他不会介意。

Jon

我首先感谢我的妻子Laura和我的孩子Michele和Giulia，在我为这个版本工作的时候，他们让我的生命充满了快乐和幸福的感觉。LWN.net的订阅者们也通过他们的慷慨和无私帮助我完成了这一工作。Linux内核开发者为我提供了很好的服务，他们让我成为他们社区的一员，回答我的问题并在我困惑时扫清障碍。我也要感谢来自世界各地读者的关于本书第二版的评论，这些评论让我高兴并受到鼓舞。我尤其要感谢Alessandro Rubini在第一版时启动这项编写工作（并一直持续到现在这个版本），还有Greg Kroah-Hartman，他为很多章节带来了相当好的编写技巧，并获得了很好的效果。

Alessandro

我要感谢促成本书的那些人。首先要感谢的是Federica，在我们的蜜月期间当我在帐篷中于笔记本电脑上审校本书第一版时，她给予了我充分的理解和支持。我还要感谢Giorgio和Giulia，他们卷入了本书后面版本的编写，并乐于接受成为一个经常开夜车的“角马（gnu）”的儿子。我还要感谢许多自由软件作者，他们让自己的作品供任何人研究，从而教会了我如何编程。对这个版本而言，我尤其感谢Jon和Greg，他们已成为整

个编写工作的主要伙伴；没有他们中的任何一个人，编写工作都不可能完成，因为代码库越来越大且越来越艰涩，而我的时间却越来越少。Jon是这一版本的主要领导，而他们两位在SMP和数字计算器上的专业知识也大大弥补了我在编程技术上的不足。

Greg

我要感谢我的妻子Shannon和我的孩子Madeline和Griffin，他们对我的工作给予了理解和忍耐。如果没有他们支持我最初在Linux上的开发，那么我根本没有机会来完成这本书的编写工作。我也要感谢Alessandro和Jon，他们为我提供了编写本书的机会，我也很荣幸能够参与其中。我还要衷心感谢Linux内核程序员，他们无私奉献的代码使得我和其他人有机会通过阅读而受益。当然，我还要感谢发送给我缺陷报告、批评我的代码、指出我的愚蠢做法的人，他们教会了我如何成为一名更好的程序员，通过这些使得我非常骄傲成为社区的一份子。感谢你们!



第一章

设备驱动程序简介

以Linux为代表的众多自由操作系统有许多优点，其中之一就是它们的内部实现细节对所有人来讲都是公开的。原先，操作系统的代码仅仅掌握在少数程序员手中，模糊而神秘，但现在任何人只要具备必要的技术能力即可方便地验证、理解和修改操作系统。在让操作系统民主化的进程中，Linux发挥了重要作用。Linux内核由大量而且复杂的代码组成，但是希望成为内核黑客的人需要一个入口，通过这个入口他们能够方便地参与到Linux内核开发而不会被内核代码的复杂性淹没。通常，设备驱动程序就是这个进入Linux内核世界的大门。

设备驱动程序在Linux内核中扮演着特殊的角色。它们是一个个独立的“黑盒子”，使某个特定硬件响应一个定义良好的内部编程接口，这些接口完全隐藏了设备的工作细节。用户的操作通过一组标准化的调用执行，而这些调用独立于特定的驱动程序。将这些调用映射到作用于实际硬件的设备特有操作上，则是设备驱动程序的任务。这个编程接口能够使得驱动程序独立于内核的其他部分而建立，必要的情况下可在运行时“插入”内核。这种模块化的特点使得Linux驱动程序的编写非常简单，因此内核驱动程序的数目也迅速增长，目前已有成百上千的驱动程序可用。

促使我们对Linux驱动程序的编写感兴趣的原因有很多。首先，仅新硬件问世（或过时）的速度就会使驱动程序编写人员面临很多任务；其次，个人用户可能需要了解一些驱动程序知识才能访问设备；另外，硬件厂商通过提供Linux驱动程序能为自己的产品带来数目庞大且日益增长的潜在用户群；最后，Linux系统是开源的，如果驱动程序作者愿意，驱动程序源码就可以在大量用户中间迅速流传。

本书将讲述有关驱动程序编程方法以及内核的相关知识。我们尽量采取独立于硬件的方法，所讲述的编程技巧和接口尽可能不依赖于任何具体设备。每个驱动程序都不尽相同，作为驱动程序开发者，我们应该很好地了解自己面对的具体设备。但是，驱动程序相关

的大部分原理和技巧都是相同的。本书不准备讲述具体的设备，而主要集中在让设备工作的背景知识上。

刚开始学习编写驱动程序时，会经常碰到许多关于Linux内核的知识；它将帮助我们理解机器如何工作，工作为什么不像预期的那样快，或者为什么没有产生预期的结果等等。我们将逐步地介绍新知识，先从简单的驱动程序开始，然后逐步构造复杂的驱动程序。每个新概念都带有示例代码，它们不需要特别的硬件支持就可以运行。

本章不涉及实际的编程。但我们也会介绍一些有关Linux内核的背景知识，这些知识在后面进行实际编程时将非常有用。

设备驱动程序的作用

作为驱动程序编写者，我们需要在所需的编程时间以及驱动程序的灵活性之间选择一个可接受的折衷。读者可能奇怪于说驱动程序“灵活”，我们用这个词实际上是强调设备驱动程序的作用在于提供机制，而不是提供策略。

区分机制和策略是Unix设计背后隐含的最好思想之一。大多数编程问题实际上都可以分成两部分：“需要提供什么功能”（机制）和“如何使用这些功能”（策略）。如果这两个问题由程序的不同部分来处理，或者甚至由不同的程序来处理，则这个软件包更易开发，也更容易根据需要来调整。

例如，Unix中图形显示器的管理就分成X服务器以及窗口和会话管理器两部分。前者操作硬件，给用户程序提供统一接口；后者实现特定策略，而不用知道任何与硬件相关的信息。我们可以在不同硬件上运行同样的窗口管理器，不同的用户也可以在相同的工作站上使用不同的配置。即使完全不同的桌面环境，诸如KDE和GNOME，也能在同一个系统中共存。另外一个例子是具有分层结构的TCP/IP网络：位于下层的操作系统负责提供套接字抽象层，但在所传输的数据上则没有附加任何策略；上面各层的服务器则分别提供不同的服务（以及相关策略）。再比如，一个类似*ftpd*这样的服务器提供文件传输机制，用户可以使用任何自己喜欢的客户端传输文件，例如命令行和图形客户端；而人们也可以编写一个新的用户界面来传输文件。

驱动程序同样存在机制和策略的分离问题。例如，软驱的驱动程序不带策略，它的作用是将磁盘表示为一个连续的数据块阵列。系统高层负责提供策略，比如谁有权访问软盘驱动器，是直接访问驱动器还是通过文件系统，以及用户是否可以在驱动器上挂装文件

系统等等。既然不同的环境通常需要不同的方式来使用硬件，我们应当尽可能做到让驱动程序不带策略。

在编写驱动程序时，程序员应该特别注意下面这个基本概念：编写访问硬件的内核代码时，不要给用户强加任何特定策略。因为不同的用户有不同的需求，驱动程序应该处理如何使硬件可用的问题，而将怎样使用硬件的问题留给上层应用程序。因此，当驱动程序只提供了访问硬件的功能而没有附加任何限制时，这个驱动程序就比较灵活。然而，有时候我们也需要在驱动程序中实现一些策略。例如，某个数字I/O驱动程序只提供以字节为单位访问硬件的方法，这样就可避免编写额外代码来处理单个数据位的麻烦。

如果从另外一个角度来看驱动程序，它还可以看作是应用程序和实际设备之间的一个软件层。驱动程序的这种特权角色可让编写者选择如何展现设备特性，也就是说，即使对于相同的设备，不同的驱动程序可能提供不同的功能。实际的驱动程序设计应该在许多要考虑的因素之间做出平衡。例如，某个驱动程序可能同时被不同的程序并发地使用，此时驱动程序的程序员就有绝对的自由来决定如何处理并发问题：可以在设备上实现独立于硬件功能的内存映射；也可以提供一个用户函数库，以帮助应用程序开发者在原语基础上实现新的策略，等等。总地来说，驱动程序设计主要还是综合考虑下面三个方面的因素：提供给用户尽量多的选项、编写驱动程序要占用的时间以及尽量保持程序简单而不至于错误丛生。

不带策略的驱动程序包括一些典型的特征：同时支持同步和异步操作、驱动程序能够被多次打开、充分利用硬件特性，以及不具备用来“简化任务”的或提供与策略相关的软件层等。这种类型的驱动程序不仅能很好地服务最终用户，而且易于编写和维护。实际上，不带策略是软件设计者的一个共同目标。

实际上，许多设备驱动程序是同用户程序一起发行的。这些用户程序主要用来帮助配置和访问目标设备。它们可能是简单的工具，也可能是完整的图形应用程序。例如，用来调整并口打印机驱动程序工作方式的*tunelp*程序；作为PCMCIA驱动程序包一部分的图形化*cardctl*工具等等。和驱动程序一起提供的还会有一个客户程序库，它提供了那些不必在驱动程序本身实现的功能。

本书的讨论范围局限于内核，因此我们将尽量避免讨论策略、应用程序和支持库的问题。有时可能确实会涉及到有关策略以及如何支持策略的内容，但我们不会深入讨论使用设备的用户程序及它们所实现的策略。另外，我们应该清楚，用户程序是软件包的有机组成部分，即使不带策略的软件包，也会同时发布配置文件为下层机制提供默认配置。

内核功能划分

Unix系统支持多个进程的并发运行，每个进程都请求系统资源，比如运算、内存、网络连接或其他一些资源等。内核负责处理所有这些请求，根据内核完成任务的不同（这些任务之间的区别可能不总是那么清楚），如图1-1所示，可将内核功能分成如下几部分：

进程管理

进程管理功能负责创建和销毁进程，并处理它们和外部世界之间的连接（输入输出）。不同进程之间的通信（通过信号、管道或进程间通信原语）是整个系统的基本功能，因此也由内核处理。除此之外，控制进程如何共享CPU的调度器也是进程管理的一部分。概括来说，内核进程管理活动就是在单个或多个CPU上实现了多个进程的抽象。

内存管理

内存是计算机的主要资源之一，用来管理内存的策略是决定系统性能的一个关键因素。内核在有限的可用资源之上为每个进程都创建了一个虚拟地址空间。内核的不同部分在和内存管理子系统交互时使用一组函数调用，包括简单的*malloc*/*free*函数对以及其他一些复杂的函数。

文件系统

Unix在很大程度上依赖于文件系统的概念，Unix中的每个对象几乎都可以当作文件来看待。内核在没有结构的硬件上构造结构化的文件系统，而文件抽象在整个系统中广泛使用。另外，Linux支持多种文件系统类型，也就是在物理介质上组织数据的不同方式。例如，磁盘可以格式化为符合Linux标准的ext3文件系统，也可格式化为常用的FAT文件系统或者其他种类。

设备控制

几乎每一个系统操作最终都会映射到物理设备上。除了处理器、内存以及其他很有限的几个对象外，所有设备控制操作都由与被控制设备相关的代码来完成，这段代码就叫做驱动程序。内核必须为系统中的每件外设嵌入相应的驱动程序，这包括硬盘驱动器、键盘和磁带驱动器等。这方面的内核功能将是本书讨论的主题。

网络功能

网络功能也必须由操作系统来管理，因为大部分网络操作和具体进程无关：数据包的传入是异步事件。在某个进程处理这些数据包之前必须收集、标识和分发这些数据包。系统负责在应用程序和网络接口之间传递数据包，并根据网络活动控制程序的执行。另外，所有的路由和地址解析问题都由内核处理。

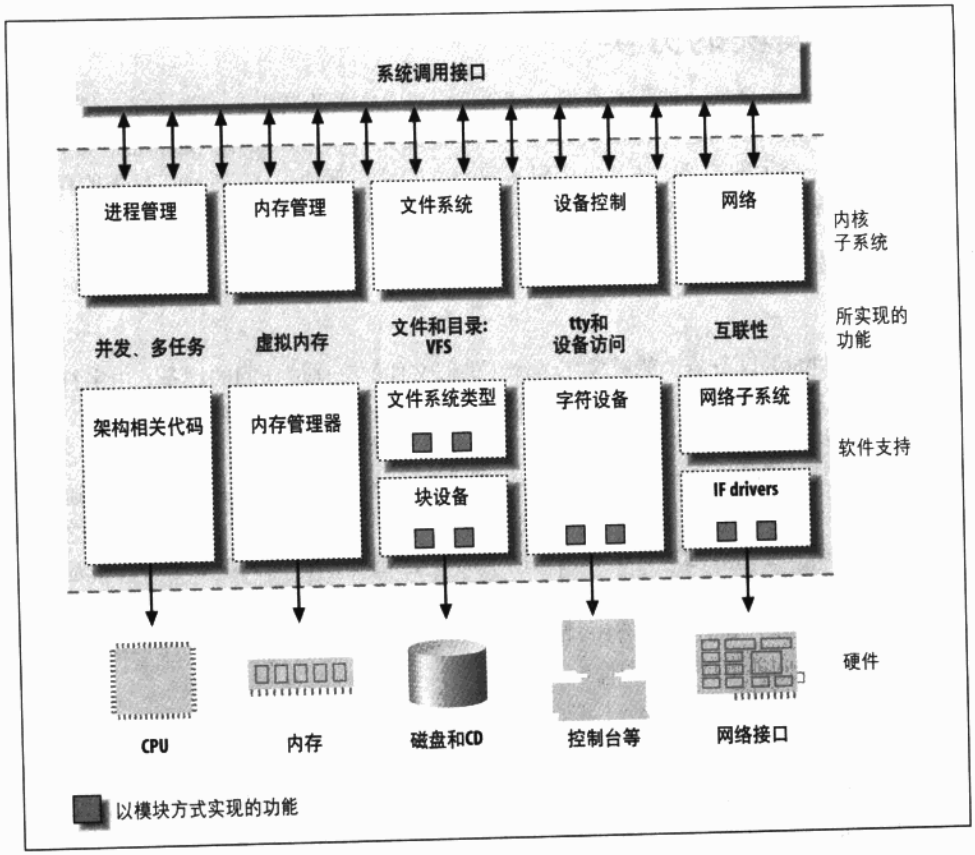


图 1-1：内核功能的划分

可装载模块

Linux 有一个很好的特性：内核提供的特性可在运行时进行扩展。这意味着当系统启动并运行时，我们可以向内核添加功能（当然也可以移除功能）。

可在运行时添加到内核中的代码被称为“模块”。Linux 内核支持好几种模块类型（或者类），包括但不限于设备驱动程序。每个模块由目标代码组成（没有连接成一个完整的可执行程序），我们可以使用 *insmod* 程序将模块连接到正在运行的内核，也可以使用 *rmmod* 程序移除连接。

图 1-1 标识了负责特定任务的几个不同的模块类。我们根据模块提供的功能将其划分为不同的类。图 1-1 中的模块涵盖了几个最重要的模块类，但远远不是完全的模块类，因为在 Linux 中越来越多的功能正在被模块化。

设备和模块的分类

Linux系统将设备分成三种基本类型，每个模块通常实现为其中某一类：字符模块、块模块或网络模块。然而这种将模块分成不同类型或类的分类方式并不是非常严格，程序员可以构造一个大的模块，在其中实现不同类型的设备驱动程序。然而，优秀的程序员通常还是为每个新功能创建一个不同的模块，从而实现良好的伸缩性和扩展性。

这三种类型如下：

字符设备

字符（char）设备是个能够像字节流（类似文件）一样被访问的设备，由字符设备驱动程序来实现这种特性。字符设备驱动程序通常至少要实现open、close、read和*write*系统调用。字符终端（*/dev/console*）和串口（*/dev/ttys0*以及类似设备）就是两个字符设备，它们能够很好地说明“流”这种抽象概念。字符设备可以通过文件系统节点来访问，比如*/dev/tty1*和*/dev/lp0*等。这些设备文件和普通文件之间的唯一差别在于对普通文件的访问可以前后移动访问位置，而大多数字符设备是一个个只能顺序访问的数据通道。然而，也存在具有数据区特性的字符设备，访问它们时可前后移动访问位置。例如，帧抓取器就是这样一个设备，应用程序可以用*mmap*或*lseek*访问抓取的整个图像。

块设备

和字符设备类似，块设备也是通过*/dev*目录下的文件系统节点来访问。块设备（例如磁盘）上能够容纳文件系统。在大多数Unix系统中，进行I/O操作时块设备每次只能传输一个或多个完整的块，而每块包含512字节（或2的更高次幂字节的数据）。Linux可以让应用程序像字符设备一样地读写块设备，允许一次传递任意多字节的数据。因而，块设备和字符设备的区别仅仅在于内核内部管理数据的方式，也就是内核及驱动程序之间的软件接口，而这些不同对用户来讲是透明的。在内核中，和字符驱动程序相比，块驱动程序具有完全不同的接口。

网络接口

任何网络事务都经过一个网络接口形成，即一个能够和其他主机交换数据的设备。通常，接口是个硬件设备，但也可能是个纯软件设备，比如回环（loopback）接口。网络接口由内核中的网络子系统驱动，负责发送和接收数据包，但它不需要了解每项事务如何映射到实际传送的数据包。许多网络连接（尤其是使用TCP协议的连接）是面向流的，但网络设备却围绕数据包的传输和接收而设计。网络驱动程序不需要知道各个连接的相关信息，它只要处理数据包即可。

由于不是面向流的设备，因此将网络接口映射到文件系统中的节点（比如*/dev/tty1*）比较困难。Unix访问网络接口的方法仍然是给它们分配一个唯一的名字（比

如eth0)，但这个名字在文件系统中不存在对应的节点。内核和网络设备驱动程序间的通信，完全不同于内核和字符以及块驱动程序之间的通信，内核调用一套和数据包传输相关的函数而不是 *read*、*write* 等。

还有另外一种划分驱动程序模块类型的方法。一般而言，某些驱动程序类型同内核用来支持某种给定类型设备的附加层一起工作。比如，通用串行总线（USB）模块、串行模块、SCSI 模块，等等。每个 USB 设备由一个 USB 模块驱动，而该 USB 模块和 USB 子系统一同工作，但设备本身在系统中表现为一个字符设备（比如 USB 串口）、一个块设备（比如 USB 存储卡读取器），或者一个网络设备（比如 USB 以太网接口）。

最近还有其他类型的设备驱动程序加入到了内核，其中包括 FireWire 驱动程序和 I2C 驱动程序。与处理 USB 及 SCSI 驱动程序的方法一样，内核开发者实现整个设备类型的共有特性，然后提供给驱动程序实现者，从而避免了重复工作，降低了出现缺陷的可能，简化并增强了编写这些驱动程序的过程。

除了设备驱动程序外，内核中其他一些功能（不管是硬件还是软件功能）也都模块化了。一个常见的例子是文件系统。一个文件系统类型决定了如何在块设备上组织数据，以表示目录和文件形成的树。文件系统并不是设备驱动程序，因为没有任何实际物理设备同这种信息组织方式相关联。相反，文件系统类型是个软件驱动程序，它将低层数据结构映射到高层数据结构，决定文件名可以有多长以及在目录项中存储文件的哪些信息等等。文件系统模块必须实现访问目录和文件的最底层系统调用，方法是将文件名和路径（以及其他一些信息，比如访问模式等）映射到数据块中的数据结构中。这种接口完全独立于在磁盘（或其他介质）上传输的实际数据，而数据的传输由块设备驱动程序负责完成。

由于 Unix 系统严重依赖于底层的文件系统，因此文件系统概念对系统操作具有重要意义。解释（特定）文件系统信息的功能位于内核层次结构的最底层，具有极其重要的作用。如果我们想为一款新的 CD-ROM 编写块驱动程序，则必须提供对 CD-ROM 上包含的数据进行 *ls* 或 *cp* 等操作的功能，否则驱动程序毫无用处。Linux 支持文件系统模块的概念，它的软件接口声明了可以在文件系统中的节点、目录、文件以及超级块上执行的不同操作。不过，程序员需要自己编写文件系统模块的情况比较少见，因为正式发行的内核版本中已经包含了最重要文件系统类型的代码。

安全问题

安全问题在当今社会日益引起人们的关注，本书在适当的时候都会讨论这一问题。然而，有必要现在就弄清楚几个原则性的概念。

系统中的所有安全检查都是由内核代码进行的，如果内核有安全漏洞，则整个系统就会有安全漏洞。在正式发行的内核版本中，只有授权用户才能装载模块；也就是说，系统调用*init_module*会检查调用进程是否具有将模块装载到内核的权利。因此，运行正式发布的内核时，只有超级用户（注1）或者成功成为超级用户的入侵者才能使用特权代码。

驱动程序编写者应当尽量避免在代码中实现安全策略。安全策略问题最好在系统管理员的控制之下，由内核的高层来实现。当然也会有例外。作为驱动程序编写者，我们应当清楚有些情况下，某些种类的设备访问会影响整个系统，因此应该适当控制。例如，能够影响全局资源的设备操作（比如设置中断线），可能会破坏硬件（比如装载固件）或者影响其他用户（比如给磁带驱动器设置默认的块尺寸），因此通常只能由特权用户执行，而相关的安全检查必须由驱动程序本身完成。

当然，驱动程序编写者还应当避免由于自身原因引入安全方面的缺陷。C编程语言很容易产生几种类型的错误，比如1缓冲区溢出就会导致许多安全问题。缓冲区溢出通常是由于程序员忘记检查缓冲1区中已写入了多少数据，导致数据写到了缓冲区边界之外，从而覆盖了系统中的其他数据。这种错误可能危及整个系统的安全，因此必须尽量避免。幸运的是，在驱动程序环境中避免这种错误通常相对容易，因为此时的用户接口比较有限而且经过了较为严格的控制。

还有其他一些原则性的安全概念值得注意。任何从用户进程得到的输入只有经过内核严格验证后才能使用。我们还必须小心对待未初始化的内存：任何从内核中得到的内存，都必须在提供给用户进程或者设备之前清零或者以其他方式初始化，否则就可能发生信息泄漏（如数据和密码的泄漏等）。如果设备要解释和分析发送给它的数据，则必须确保用户不能将可能破坏系统的任何东西发送给它。最后，我们还应当考虑设备操作可能造成的影响；如果某些特定操作（比如重新装载适配卡上的固件或者格式化磁盘）可能会影响整个系统，则应当将此类操作限于特权用户。

应当小心使用从第三方获得的软件，特别是与内核相关时更是如此，这是因为源码是开放的，每个人都可以修改和重新编译它。尽管通常我们可以信任发行版本中预先编译的内核，但当使用由一个我们不是非常熟悉的朋友编译的内核时就得当心——就像我们不愿意以root身份运行一个预先编译的二进制文件一样，我们也不应当运行一个预先编译好的内核。例如，一个恶意修改过的内核可能会允许任何人装载模块，这样，一扇通过*init_module*的后门就打开了。

注1：从技术上讲，只有个别有CAP_SYS_MIDULE能力的人可以执行此操作。我们将在第六章对此进行讨论。

Linux内核也可编译为不支持模块方式，从而可以关闭任何模块相关的安全漏洞。但在这种情况下，所有所需的驱动程序必须直接编译到内核中。另外，在2.2及以后的内核版本中，我们还可以通过权能机制禁止在系统启动后装载内核模块。

版本编号

在深入探讨编程之前，我们还要说明一下Linux使用的版本编号机制以及本书讲到的内核版本。

首先，Linux系统中的每个软件包都有自己的发行编号，而且它们之间经常存在相互间的依赖关系，也就是说，只有存在某个软件包的某个特定版本时，才能运行另一个软件包的某个特定版本。通常，Linux发行版的制作者已经解决了复杂的包匹配问题，用户安装一个预先打包好的发行版时不必关心版本号问题。但如果我们需要自己替换或者更新系统中的某个软件包，则另当别论。幸运的是，现在几乎所有的发行版都带有包管理器，它在验证满足包之间的依赖关系后才允许升级包。

为了运行本书中的示例代码，除了内核的2.6版本之外，对其他工具则没有版本要求。任何最近发行的Linux发行版都可以用来运行我们的例子。我们不会详细阐述具体的要求，因为读者遇到任何版本相关的问题时，均可参考内核源文件*Documentation/Changes*来解决。

对内核来讲，偶数编号的内核版本（如2.6.*x*）是用于正式发行的稳定版本，而奇数编号的版本（如2.7.*x*）则是开发过程中的一个快照，它将很快被下一开发版本更新。最新的开发版本只是代表了内核开发目前的状态，几天后可能就会过时。

本书描述内核的2.6版本。我们的关注点主要在于向读者展示2.6.10内核中有关设备驱动程序编写的所有可用功能特性。2.6.10是编写本书时的最新版本。与先前的版本不同，本书的这个版本不再讨论老的内核版本。如果读者对老的版本感兴趣，可阅读本书第二版，该版本描述了内核的2.0到2.4版本，而且可在线获得：*http://lwn.net/Kernel/LDD2/*。

内核程序员需要注意内核的开发过程在2.6版本中有所变化。2.6系列目前正在接受一些修改，在以前这些修改可能会被认为对一个“稳定”内核来说显得太大。这种情况意味着内核的内部编程接口可能发生变化，从而会让本书的一些内容变得陈旧。出于对此原因的考虑，文中随附的示例代码可在2.6.10版本下工作，而某些模块不能在较早的版本中编译。我们鼓励那些希望跟进内核改变的程序员加入本书参考书目中列出的邮件列表，或者访问列出的网站。在网址*http://lwn.net/Articles/2.6-kernel-api/*中包含有自从本书发行以来内核API所发生的一些变化。

本书很少讨论到奇数编号的内核版本，普通用户也很少会有使用这种版本的需求。然而，如果我们希望了解、跟踪开发版本的新特性，就需要运行最近发行的开发版本，而且还得随着开发版本的更新不断获取缺陷的补丁以及新实现的特性。但是，对于开发版本我们必须记住它没有任何担保（注2），如果我们碰到的问题是由于老版本奇数编号的内核引起的，则没有人可以求助。那些运行奇数编号内核版本的程序员通常具有足够的知识，无需求助教科书就可以自己钻研内核代码。这也是我们为什么不在这儿讨论内核开发版本的另外一个原因。

Linux的另外一个特性是，它是一个不依赖于特定硬件平台的操作系统。目前，它不再是“PC克隆上的Unix克隆”，而支持20多种架构。本书尽可能做到与平台无关，所有示例代码都在x86和x86-64平台上测试过了。因为示例代码在32位和64位处理器上都经过了测试，所以在其他平台上应该都能编译运行。然而，如果示例代码依赖于某个特定硬件，则不能在所有已支持的平台上都能工作，我们会在源码中特别声明这一点。

许可证条款

Linux遵循GNU通用公共许可证（General Public License，GPL）版本2发布。GPL由自由软件基金会为GNU项目设计，它允许任何人重新发行甚至销售由GPL条款保护的产品，前提是产品接收者能够获得源码并拥有同样的权利。另外，任何从GPL保护的产品中派生出来的软件产品也必须在GPL条款下发布。

这样一个许可证的主要目的是通过允许每个人自由修改程序来实现知识增长；同时，向公众出售软件的人仍旧可以获利。但就是这样一个具有简单目的的条款，在GPL及其使用上一直存在着争论。如果读者想阅读这个许可证的原文，可以在系统的好几个地方找到它，比如内核源代码树顶层目录中的*COPYING*文件。

生产商经常询问他们能否仅以二进制形式发布内核模块。这个问题的答案被有意地保持为不确定。到目前为止，只要二进制模块只使用公开的内核接口，则二进制形式的发布得以容忍。但内核版权由许多开发人员拥有，并不是所有的人都同意内核模块不算作派生产品。如果你或者你的雇主打算在非自由的许可证下发布内核模块，则应该和你的法律顾问认真讨论是否可行。同时也请注意，内核开发者不会考虑因为内核版本间（甚至稳定的内核版本间）的变化而导致二进制模块出现不兼容的问题。如果可能，你应该以自由软件的形式发布你的模块，这样有利于你和你的用户。

注2：　注意，即使是偶数编号的内核也一样没有担保，除非依赖于由供应商提供担保的版本。

如果你希望自己的代码进入内核的主分支，或者你的代码需要向内核中打补丁，则必须在发布代码时使用GPL兼容的许可证。尽管对你代码的个人使用并不强制要求GPL，但如果要发布你的代码，则必须同时发布源代码，也就是说，获得你所发布软件包的人必须被允许重新建立对应的二进制代码。

就本书而言，不管是源代码还是二进制形式，文中的大部分代码都可以免费重新发布，并且不管是作者还是O'Reilly都不对任何派生作品保留任何权利。所有程序都可以从*ftp://ftp.ora.com/pub/examples/linux/drivers/*中得到，许可证条款在同一目录下的LICENSE文件中表述。

加入内核开发社团

当我们开始为Linux内核编写模块的时候，我们就成为巨大开发者社区中的一员了。在这个社区中，我们不仅发现很多人从事类似的工作，而且发现一群具有高度使命感的工程师正朝着将Linux发展成为一个更好系统的目标前进。这些人是我们获得帮助、思路以及严格评价的源泉。当我们为新的驱动程序寻找测试者的时候，他们将是我们乐于提交的第一批人。

*linux-kernel*邮件列表是Linux内核开发者的聚集中心。从Linus Torvalds往下，所有主要的内核开发者都订阅这个邮件列表。请注意，这个列表不适合那些心脏比较脆弱的人：该邮件列表每天都会有200条消息或者更多。然而，对于那些对内核开发感兴趣的人来说，跟踪这个列表是必要的；对于那些需要内核开发帮助的人来讲，它更是一个顶级质量的资源。

要加入linux-kernel列表，请遵照linux-kernel邮件列表FAQ，即*http://www.tux.org/lkml*中的指示。如果已经打开了这个FAQ页面，我们还应当看看其中的其他内容，它上面有大量的有用信息。Linux内核开发者都比较忙，他们更愿意帮助那些首先了解了基本知识的人。

本书概要

从第二章起，我们将进入内核编程领域。第二章介绍了模块化技术，解释了其实现技巧，并讲解了运行模块的代码。第三章讨论字符驱动程序，给出了一个基于内存的设备驱动程序的完整代码。将内存作为设备的硬件基础，可允许任何人在无需特殊硬件的情况下运行我们的示例代码。

对程序员来讲，调试技术是很重要的工具，我们将在第四章介绍内核调试技术。对那些希望在内核中有所作为的人来讲，并发管理和竞态也一样重要。第五章主要关注因为资源的并发访问而引入的问题，并介绍了Linux用来控制并发的机制。

随后，带着调试和并发管理技巧，我们转到字符驱动程序的高级特性，比如阻塞操作、*select*的使用以及重要的*ioctl*调用等，这些都是第六章的内容。

在讨论硬件管理之前，我们先剖析内核的几个软件接口：第七章讨论内核的时间管理，第八章讨论内存分配。

接下来我们集中于硬件问题。第九章描述I/O端口管理和设备上内存缓冲区的管理。之后，我们在第十章讨论中断处理。不幸的是，并不是所有人都可以运行这些章节中的示例代码，因为需要一些硬件支持才能测试软件接口中断。我们尽可能使必需的硬件支持减到最小，但仍然需要一些简单的硬件来运行这些章节中的示例代码，比如一个标准的并口。

第十一章介绍了内核数据类型的使用，以及可移植代码的编写。

本书的第二部分专门用来讨论更高级的主题。我们从深入探讨硬件、尤其是一些具体总线的功能开始。第十二章介绍了编写PCI设备驱动程序的细节信息，而第十三章讲解了USB设备相关的API。

有了对外设总线的理解，我们可以详细研究Linux的设备模型了。设备模型是由内核使用的抽象层，它描述了内核所管理的硬件和软件资源。第十四章从底向上研究设备模型这种基础设施，首先讲述了kobject类型。这一章描述了设备模型和真实硬件间的集成，然后利用这些知识讲述了诸如热插拔设备和电源管理等相关内容。

在第十五章，我们转而讨论Linux的内存管理。这一章说明了如何将内核内存映射到用户空间（即*mmap*系统调用）、如何将用户内存映射到内核空间（使用*get_user_pages*），以及如何将这两种内存映射到设备空间（执行直接内存访问[DMA]操作）。

对内存的理解将有助于阅读之后的两个章节，这两章讲述了其他几个主要的驱动程序类型。第十六章介绍了块驱动程序，并说明了它和字符驱动程序之间的区别。第十七章讲述了网络驱动程序的编写。最后，我们在第十八章讨论了串行驱动程序，然后以“参考书目”结束本书。



第二章

构造和运行模块

现在终于可以开始编程了。本章将介绍所有关于模块编程和内核编程的必要概念。在这有限的篇幅中，我们将构建并运行一个完整的（但相对没有多少用处的）模块。我们会看到一些由所有模块共享的基本代码。掌握这种技能是编写任何模块化驱动程序的基础。为了避免一次引入太多概念，本章将只讨论模块，而避免涉及任何特定类型的设备。

本章引入的所有内核条目（函数、变量、头文件和宏）将在本章末尾的“快速参考”一节中集中描述。

设置测试系统

从本章开始，我们为读者提供一些示例模块，以便演示编程概念（所有的示例代码可从O'Reilly的FTP站点上下载，相关信息请参阅第一章）。构造、加载和修改这些示例代码，将帮助读者理解驱动程序的工作方式以及和内核的交互方式。

示例模块应可在几乎所有的2.6.x内核之上运行，也包括发行版厂商提供的内核。但是，我们建议读者直接从*kernel.org*的镜像网站上获得一个“主线”内核，并安装到自己的系统中。由发行版厂商提供的内核通常打了许多的补丁，从而和主线内核存在很大差异；某些情况下，厂商的补丁会修改设备驱动程序使用的内核API。如果读者正在编写一个只适用于某特定发行版的驱动程序，则应该针对相关内核创建和测试自己的驱动程序。但是，如果想要学习驱动程序的编写，则标准内核是最好的。

不管内核来自哪里，要想为2.6.x内核构造模块，还必须在自己的系统中配置并构造好内核树。这一要求和先前版本的内核不同，先前的内核只需要有一套内核头文件就够了。但因为2.6内核的模块要和内核源代码树中的目标文件连接，通过这种方式，可得到一个更加健壮的模块装载器，但也需要这些目标文件存在于内核目录树中。这样，读者首先要准备好一个内核源代码树（可以是来自kernel.org网络的，也可以是发行版的内核

源代码包），构造一个新内核，然后安装到自己的系统中。因为后面讲到的原因，如果在构造内核时运行的恰好是目标内核，则开发工作就会非常轻松。当然，这并不是必需的。

警告： 另外，读者还应该想好在什么地方完成模块的试验、开发和测试。我们已尽可能让示例模块安全而正确，但缺陷依然可能存在。内核代码中的错误可能导致用户进程甚至整个系统的崩溃。这些错误通常不会制造更加严重的问题，比如磁盘的损坏，但我们仍然建议读者在一个不包含任何敏感数据或者不执行重要服务的系统上完成我们的内核试验。内核黑客通常拥有一个“牺牲用的”系统，用于测试新的代码。

因此，如果读者还没有合适的系统，其中包含已经配置并构造好了的内核源代码树，则现在就应该准备好。一旦完成准备工作，读者就可以尝试运行内核模块了。

Hello World 模块

许多编程书籍都会以一个“hello world”示例程序来说明最简单的程序。虽然本书讲述的是内核模块而不是程序，但如果读者急于看到实际的代码，则下面这段代码是完整的“hello world”模块：

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}

static void hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

这个模块定义了两个函数，其中一个在模块被装载到内核时调用（*hello_init*），而另一个则在模块被移除时调用（*hello_exit*）。*module_init* 和 *module_exit* 行使用了内核的特殊宏来表示上述两个函数所扮演的角色。另外一个特殊宏（*MODULE_LICENSE*）用来告诉内核，该模块采用自由许可证；如果没有这样的声明，内核在装载该模块时会产生抱怨。

函数*printk*在Linux内核中定义，功能和标准C库中的函数*printf*类似。内核需要自己单独的打印输出函数，这是因为它在运行时不能依赖于C库。模块能够调用*printk*是因为在*insmod*函数装入模块后，模块就连接到了内核，因而可以访问内核的公用符号（包括函数和变量，下一节详述）。代码中的字符串KERN_ALERT定义了这条消息的优先级（注1）。我们需要在模块代码中显式地指定高优先级的原因在于：具有默认优先级的消息可能不会输出在控制台上，这依赖于内核版本、*klogd*守护进程的版本以及具体的配置。读者可以暂时忽略这个问题，我们将在第四章中仔细阐述。

如下所示，读者可以通过调用*insmod*和*rmmod*工具来测试这个模块。值得注意的是，只有超级用户才有权加载和卸载模块。

```
% make
make[1]: Entering directory `/usr/src/linux-2.6.10'
  CC [M]  /home/ldd3/src/misc-modules/hello.o
  Building modules, stage 2.
  MODPOST
  CC      /home/ldd3/src/misc-modules/hello.mod.o
  LD [M]  /home/ldd3/src/misc-modules/hello.ko
make[1]: Leaving directory `/usr/src/linux-2.6.10'
% su
root# insmod ./hello.ko
Hello, world
root# rmmod hello
Goodbye, cruel world
root#
```

需要再次提醒注意的是，为了让上述命令正常工作，读者必须已经在makefile能够找到的地方（这里是*/usr/src/linux-2.6.10*）正确配置和构造了内核树。在“编译和装载”一节中，我们将仔细描述如何构造模块。

根据系统传递消息行机制的不同，读者得到的输出结果可能不一样。需要特别指出的是，上面的屏幕输出是在文本控制台上得到的；如果读者在某个运行于Windows系统下的终端仿真器中运行*insmod*和*rmmod*，则不会在屏幕上看到任何输出。实际上，它可能输出到某个系统日志文件里，比如*/var/log/messages*（实际的名称随Linux发行版的不同可能会有所变化）。内核消息的传递机制将在第四章中详细讨论。

我们已经看到，编写一个模块并没有想像的那么困难——至少当模块不需要完成什么有价值的工作时。真正的困难在于理解设备并最大化其性能。本章将深入讨论模块化问题，而把设备相关的问题留到以后的章节。

注1：优先级只是个字符串，诸如<1>，该字符串置于*printk*格式字符串的前面。请注意，KERN_ALERT之后并不使用逗号，但添加逗号的打字错误却会经常发生，幸好编辑器能帮助我们捕获这个错误。

核心模块与应用程序的对比

在进一步讨论之前，有必要搞清楚内核模块和应用程序之间的种种不同之处。

大多数小规模及中规模应用程序是从头到尾执行单个任务，而模块却只是预先注册自己以便服务于将来的某个请求，然后它的初始化函数就立即结束。换句话说，模块初始化函数的任务就是为以后调用模块函数预先做准备；这就像模块在说："我在这儿，并且我能做这些工作。"模块的退出函数（例子中的*hello_exit*）将在模块被卸载之前调用。它告诉内核："我要离开啦，不要再让我做任何事情了。"这种编程方式和事件驱动的编程有点类似，但并不是所有的应用程序都是事件驱动的，而每个内核模块都是这样的。事件驱动的应用程序和内核代码之间的另一个主要不同是：应用程序在退出时，可以不管资源的释放或者其他的清除工作，但模块的退出函数却必须仔细撤销初始化函数所做的一切，否则，在系统重新引导之前某些东西就会残留在系统中。

顺便提及，能够卸载模块可能是模块化驱动程序编程当中读者最为喜欢的一个特色，因为它有助于缩短模块的开发周期：我们可以测试新驱动的一系列版本却不需要每次都经过冗长的关机/重启过程。

作为程序员，我们知道应用程序可以调用它并未定义的函数，这是因为连接过程能够解析外部引用从而使用适当的函数库。例如，定义在*libc*中的*printf*函数就是这种可被调用的函数之一。然而，模块仅仅被链接到内核，因此它能调用的函数仅仅是由内核导出的那些函数，而不存在任何可链接的函数库。例如，前面*hello.c*中使用的*printk*函数就是由内核定义并导出给模块使用的一个*printf*的内核版本。除了几个细小差别外，它和*printf*函数功能类似，最大的不同在于它缺乏对浮点数的支持。

图2-1展示了如何在模块中使用函数调用和函数指针，从而为运行中的内核增加新的功能。

因为没有任何函数库会和模块链接，因此，源文件中不能包含通常的头文件，像*<stdarg.h>*以及一些非常特殊的情况是仅存的例外。内核模块只能使用作为内核一部分的函数。和内核相关的任何内容都在我们安装并配置好的内核源代码树的头文件中声明，其中，大多数相关头文件保存在*include/linux*和*include/asm*目录中，但*include*的其他子目录中保存有和特定内核子系统相关的头文件。

每个内核头文件的作用将在本书中需要用到它们的时候加以介绍。

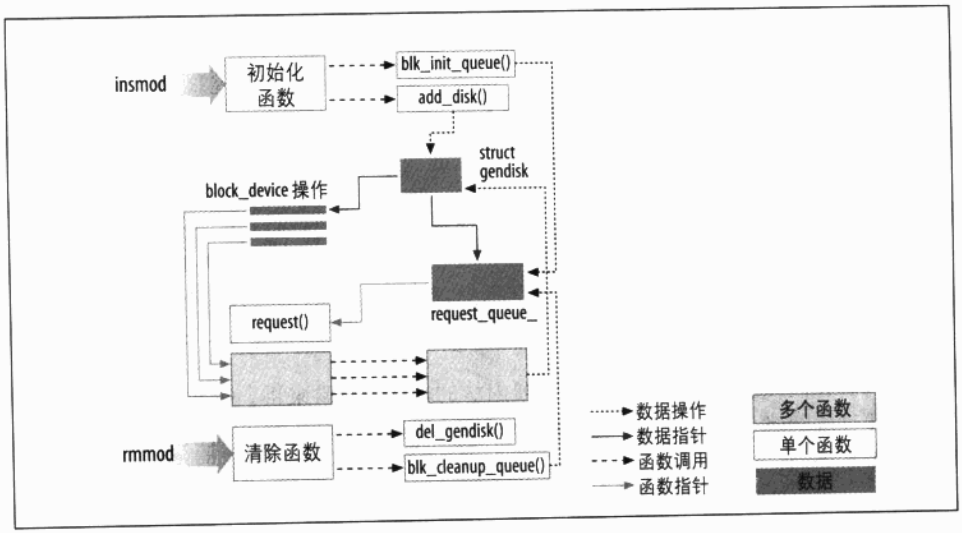


图 2-1：将模块链接到内核

内核编程和应用程序编程的另外一点重要不同之处在于各环境下处理错误的方式不同：应用程序开发过程中的段错误是无害的，并且总是可以使用调试器跟踪到源代码中的问题所在，而一个内核错误即使不影响整个系统，也至少会杀死当前进程。在第四章中，我们将看到如何跟踪内核错误。

用户空间和内核空间

模块运行在所谓的内核空间里，而应用程序运行在所谓的用户空间中。这个概念是操作系统理论的基础之一。

实际上，操作系统的作用是为应用程序提供一个对计算机硬件的一致视图。除此之外，操作系统必须负责程序的独立操作并保护资源不受非法访问。这个重要任务只有在CPU能够保护系统软件不受应用程序破坏时才能完成。

所有的现代处理器都具备这个功能。人们选择的方法是在CPU中实现不同的操作模式（或者级别）。不同的级别具有不同功能，在较低的级别中将禁止某些操作。程序代码只能通过有限数目的“门”来从一个级别切换到另一级别。Unix系统设计时利用了这种硬件特性，使用了两个这样的级别。当前所有的处理器都至少具有两个保护级别，而其他的一些处理器，比如x86系列，则有更多的级别。当处理器存在多个级别时，Unix使用最高级别和最低级别。在Unix当中，内核运行在最高级别（也称作超级用户态），在这

个级别中可以进行所有的操作。而应用程序运行在最低级别（即所谓的用户态），在这个级别中，处理器控制着对硬件的直接访问以及对内存的非授权访问。

我们通常将运行模式称作内核空间和用户空间。这两个术语不仅说明两种模式具有不同的优先权等级，而且还说明每个模式都有自己的内存映射，也即自己的地址空间。

每当应用程序执行系统调用或者被硬件中断挂起时，Unix将执行模式从用户空间切换到内核空间。执行系统调用的内核代码运行在进程上下文中，它代表调用进程执行操作，因此能够访问进程地址空间的所有数据。而处理硬件中断的内核代码和进程是异步的，与任何一个特定进程无关。

模块化代码在内核空间中运行，用于扩展内核的功能。通常来讲，一个驱动程序要执行先前讲述过的两类任务：模块中的某些函数作为系统调用的一部分而执行，而其他函数则负责中断处理。

内核中的并发

内核编程区别于常见应用程序编程的地方在于对并发的处理。大部分应用程序，除了多线程应用程序之外，通常是顺序执行的，从头到尾，而不需要关心因为其他一些事情的发生会改变它们的运行环境。内核代码并不在这样一个简单世界中运行，即使是最简单的内核模块，都需要在编写时铭记：同一时刻，可能会有许多事情正在发生。

有几方面的原因促使内核编程必须考虑并发问题。首先，Linux系统中通常正在运行多个并发进程，并且可能有多个进程同时使用我们的驱动程序。其次，大多数设备能够中断处理器，而中断处理程序异步运行，而且可能在驱动程序正试图处理其他任务时被调用。另外，有一些软件抽象（比如第七章中谈到的内核定时器）也在异步运行着。还有，Linux还可以运行在对称多处理器（Symmetric multiprocessor，SMP）系统上，因此可能同时有不止一个CPU运行我们的驱动程序。最后，在2.6中内核代码已经是可抢占的，这意味着即使在单处理器系统上也存在许多类似多处理器系统的并发问题。

结果，Linux内核代码（包括驱动程序代码）必须是可重入的，它必须能够同时运行在多个上下文中。因此，内核数据结构需要仔细设计才能保证多个线程分开执行，访问共享数据的代码也必须避免破坏共享数据。要编写能够处理并发问题而同时避免竞态（不同的执行顺序导致不同的、非预期行为发生的情况）的代码，需要一些技巧和细致的思考。对编写正确的内核代码来说，优良的并发管理是必需的；为此，本书中的示例驱动程序在编写时都考虑到了并发问题。在讲到这些驱动程序时，我们将具体介绍所使用的技术。本书第五章还会专门讨论并发问题以及内核中用于并发管理的原语。

驱动程序编写人员所犯的一个常见错误是，认为只要某段代码没有进入睡眠状态（或者阻塞），就不会产生并发问题。但即使在先前的非抢占式内核中，这种假定也是错误的。在2.6中，内核代码（几乎）始终不能假定在给定代码段中能够独占处理器。如果在编写代码时我们不注意并发问题，将可能导致出现很难调试的灾难性错误。

当前进程

虽然内核模块不像应用程序那样顺序地执行，然而内核执行的大多数操作还是和某个特定的进程相关。内核代码可通过访问全局项current来获得当前进程。current在*<asm.current.h>*中定义，是一个指向struct task_struct的指针，而task_struct结构在*<linux/sched.h>*文件中定义。current指针指向当前正在运行的进程。在*open*、*read*等系统调用的执行过程中，当前进程指的是调用这些系统调用的进程。如果需要，内核代码可以通过current获得与当前进程相关的信息，在第六章中将会介绍这样一个例子。

实际上，与早期Linux内核版本不同，2.6中current不再是一个全局变量。为了支持SMP系统，内核开发者设计了一种能找到运行在相关CPU上的当前进程的机制。这种机制必须是快速的，因为对current的引用会频繁发生。这样，一种不依赖于特定架构的机制通常是，将指向task_struct结构的指针隐藏在内核栈中。这种实现的细节同样也对其他内核子系统隐藏，设备驱动程序只要包含*<linux/sched.h>*头文件即可引用当前进程。例如，下面的语句通过访问Struct task_struct的某些成员来打印当前进程的进程ID和命令名：

```
printk(KERN_INFO "The process is \"%s\" (pid %i)\n",
        current->comm, current->pid);
```

存储在current->comm成员中的命令名是当前进程所执行的程序文件的基本名称(base name)，如果必要，会裁剪到15个字符以内。

其他一些细节

内核编程在许多方面区别于用户空间的编程，我们将在本书中逐步讨论这些区别。有一些基本的问题需要在这里说明一下，尽管没有专门的章节对这些问题进行讨论，但仍值得一提。另外，在读者深入到内核的同时，还应该时刻牢记下面讲到的这些问题。

应用程序在虚拟内存中布局，并具有一块很大的栈空间。当然，栈是用来保存函数调用历史以及当前活动函数中的自动变量的。而相反的是，内核具有非常小的栈，它可能只和一个4096字节大小的页那样小。我们自己的函数必须和整个内核空间调用链一同共享

这个栈。因此，声明大的自动变量并不是一个好主意，如果我们需要大的结构，则应该在调用时动态分配该结构。

读者经常会在内核API中看到具有两个下划线前缀（__）的函数名称。具有这种名称的函数通常是接口的底层组件，应谨慎使用。实质上，双下划线告诉程序员："谨慎调用，否则后果自负。"

内核代码不能实现浮点数运算。如果打开了浮点支持，在某些架构上，需要在进入和退出内核空间时保存和恢复浮点处理器的状态。这种额外的开销没有任何价值，内核代码中也不需要浮点运算。

编译和装载

本章开始处的"hello world"示例说明了构造模块并将其装载入系统的简单演示。当然，整个过程还需要进一步向读者说明。这一小节将详细介绍模块作者如何将源代码编译成能够装载到内核中的可执行模块。

编译模块

首先，我们要简单看看模块是如何构造的。模块的构造过程和用户空间应用程序的构造过程有很大的不同。内核是一个大的、独立的程序，为了将它的各个片断放在一起，要满足很多详细而明确的要求。和先前的内核版本相比，构造过程也有所不同；新的构造系统用起来更加简单，并可产生更加正确的结果，但看起来和先前的方法有很大的不同。内核的构造系统是个复杂的"野兽"，我们看到的只是其中之一小部分。如果读者希望理解这些表面现象之下的所有细节，则应该阅读内核源代码中*Documentation/kbuild*目录下的文件。

在构造内核模块之前，有一些先决条件首先应该得到满足。首先，读者应确保具备了正确版本的编译器、模块工具和其他必要的工具。内核文档目录中的*Documentation/Changes*文件列出了需要的工具版本；在开始构造模块之前，读者需要查看该文件并确保已安装了正确的工具。如果利用错误的工具版本来构造内核（及其模块），将导致许多细微的、复杂的问题。另外还需注意，和使用老工具一样，使用太新的工具也偶尔会导致问题；内核源代码对编译器作了大量假定，因此新的编译器版本可能导致问题的出现。

如果读者尚未准备内核树，或者尚未配置并构造内核，则应该首先完成这些工作。如果在自己的文件系统中没有2.6内核树，则无法构造可装载的模块。另外，尽管并不是必需的，但最好运行和模块对应的内核。

在准备好这些东西后，为自己的模块创建makefile则非常简单。实际上，对本章先前给出的“hello world”示例来说，下面一行就足以了：

```
obj-m := hello.o
```

如果读者熟悉*make*但对2.6内核构造系统还不熟悉的话，则可能会对此makefile的工作方式感到疑惑。毕竟上面这行并不是makefile文件的常见形式。问题的答案当然是内核构造系统处理了其余的问题。上面的赋值语句（它利用了GNU *make*的扩展语法）说明了有一个模块需要从目标文件*hello.o*中构造，而从该目标文件中构造的模块名称为*hello.ko*。

如果我们要构造的模块名称为*module.ko*，并由两个源文件生成（比如*file1.c*和*file2.c*），则正确的makefile可如下编写：

```
obj-m := module.o
module-objs := file1.o file2.o
```

为了让上面这种类型的makefile文件正常工作，必须在大的内核构造系统环境中调用它们。如果读者的内核源代码树保存在*~/kernel-2.6*目录中，则用来构造模块的*make*命令应该是（在包含模块源代码和makefile的目录中键入）：

```
make -C ~/kernel-2.6 M=`pwd` modules
```

上述命令首先改变目录到-C选项指定的位置（即内核源代码目录），其中保存有内核的顶层makefile文件。M=选项让该makefile在构造modules目标之前返回到模块源代码目录。然后，modueles目标指向obj-m变量中设定的模块；在上面的例子中，我们将该变量设置成了*module.o*。

上面这样的*make*命令还是有些烦人，因此内核开发者又开发了一种makefile方法，这种方法将使得内核树之外的模块构造变得更加容易。其技巧是用下面的方法来编写makefile：

```
# 如果已定义 KERNELRELEASE，则说明是从内核构造系统调用的，
# 因此可利用其内建语句。
ifneq ($(KERNELRELEASE),)
    obj-m := hello.o

# 否则，是直接从命令行调用的，
# 这时要调用内核构造系统。
else

    KERNELDIR ?= /lib/modules/$(shell uname -r)/build
    PWD   := $(shell pwd)
```

```
default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules

endif
```

这次，我们又看到了扩展GNU *make* 语法。在一个典型的构造过程中，该makefile将被读取两次。当makefile从命令行调用时，它注意到KENRELRELEASE变量尚未设置。我们可以注意到，已安装的模块目录中存在一个符号链接，它指向内核的构造树，这样这个makefile就可以定位内核的源代码目录。如果读者实际运行的内核并不是要构造的内核，则可以在命令行提供KERNELDIR=选项或者设置KERNELDIR环境变量，也可以修改用来设置KERNELDIR的行。在找到内核源代码树之后，这个makefile会调用default:目标，这个目标使用先前描述过的方法第二次运行*make*命令（注意，在这个makefile中*make*命令被参数化成了$(MAKE)），以便运行内核构造系统。在第二次读取该makefile文件时，它设置了obj-m，而内核的makefile负责真正构造模块。

这种构造模块的机制也许会因为其笨拙或晦涩的特点而打击读者。但是，一旦我们使用这种机制，则会欣赏内核构造系统带给我们的便利。需要注意的是，上面的makefile并不完整；一个真正的makefile应该包含通常用来清除无用文件的目标、安装模块的目标等等。读者可以在示例源代码目录中看到完整的makefile文件。

装载和卸载模块

在构造模块之后，下一步就是将模块装入内核。如前所述，*insmod*为我们完成这项工作。*insmod*程序和*ld*有些类似，它将模块的代码和数据装入内核，然后使用内核的符号表解析模块中任何未解析的符号。然而，与链接器不同，内核不会修改模块的磁盘文件，而仅仅修改内存中的副本。*insmod*可以接受一些命令行选项（参见它的手册页），并且可以在模块链接到内核之前给模块中的整型和字符串型变量赋值。因此，一个良好设计的模块可以在装载时进行配置，这比编译时的配置为用户提供了更多的灵活性，但有些情况下仍然要使用编译时的配置。本章后面的“模块参数”一节中会介绍装载时的配置方法。

感兴趣的读者可能想知道内核是如何支持*insmod*工作的，实际上它依赖于定义在*kernel/module.c*中的一个系统调用。函数*sys_init_module*给模块分配内核内存（函数*vmalloc*负责内存分配，详见第八章的“vmalloc及其相关函数”）以便装载模块，然后，该系统调用将模块正文复制到内存区域，并通过内核符号表解析模块中的内核引用，最后调用模块的初始化函数。

如果仔细阅读内核源码，我们会发现有且只有系统调用的名字前带有sys_前缀，而其

他任何函数都没有这个前缀。这种命名上的区别使我们在源码中grep系统调用时非常方便。

我们还需要进一步了解一下*modprobe*工具。和*insmod*类似，*modprobe*也用来将模块装载到内核中。它和*insmod*的区别在于，它会考虑要装载的模块是否引用了一些当前内核不存在的符号。如果有这类引用，*modprobe*会在当前模块搜索路径中查找定义了这些符号的其他模块。如果*modprobe*找到了这些模块（即要装载的模块所依赖的模块），它会同时将这些模块装载到内核。如果在这种情况下使用*insmod*，则该命令会失败，并在系统日志文件中记录“unresolved symbols（未解析的符号）”消息。

前面提到，我们可以使用*rmmod*工具从内核中移除模块。注意，如果内核认为模块仍然在使用状态（例如，某个程序正打开由该模块导出的设备文件），或者内核被配置为禁止移除模块，则无法移除该模块。配置内核并使得内核在模块忙的时候仍能“强制”移除模块也是可能的。但是，如果读者在某种情况下希望利用这种特性，则重新引导系统可能是更加合适的做法。

*lsmod*程序列出当前装载到内核中的所有模块，还提供了其他一些信息，比如其他模块是不是在使用某个特定模块等。*lsmod*通过读取*/proc/modules*虚拟文件来获得这些信息。有关当前已装载模块的信息也可以在sysfs虚拟文件系统的*/sys/module*下找到。

版本依赖

要记住，在缺少modversions的情况下，我们的模块代码必须针对要链接的每个版本的内核重新编译。我们不在这里讨论modversions，因为和开发者相比，它对发行版制作者来讲更重要些。模块和特定内核版本定义的数据结构和函数原型紧密关联，这样，一个模块看到的接口可能从一个版本到另一个版本发生重大的变化。当然，这种情况对开发中的内核来讲更是如此。

内核不会假定一个给定的模块是针对正确的内核版本构造的。我们在构造过程中，可以将自己的模块和当前内核树中的一个文件（即*vermagic.o*）链接；该目标文件包含了大量有关内核的信息，包括目标内核版本、编译器版本以及一些重要配置变量的设置。在试图装载模块时，这些信息可用来检查模块和正在运行的内核的兼容性。如果有任何不匹配，就不会装载该模块，同时可以看到如下信息：

```
# insmod hello.ko
Error inserting './hello.ko': -1 Invalid module format
```

查看系统日志文件（*/var/log/messages*或者系统配置使用的文件），将看到导致模块装载失败的具体原因。

如果读者要为某个特定的内核版本编译模块，则需要该特定版本对应的构造系统和源代码树。对前面示例 makefile 中 KERNELDIR 变量的简单修改可以实现这个目的。

在不同的发布之间，内核接口经常会发生变化。如果读者打算编写一个能够和多个内核版本一起工作的模块（尤其是必须跨主发行号工作），则必须使用宏以及 #ifdef 来构造并编译自己的代码。本书的这一版本仅仅关心内核的一个主发行号，因此读者不会在示例代码中经常看到版本测试的相关代码。但是这种需求还是会偶尔出现。在这种情况下，读者可使用 *linux/version.h* 中的相关定义。这个头文件自动包含于 *linux/module.h*，并定义了下面这些宏：

UTS_RELEASE

宏 UTS_RELEASE 扩展为一个描述内核版本的字符串，例如 "2.6.10"。

LINUX_VERSION_CODE

宏 LINUX_VERSION_CODE 扩展为内核版本的二进制表示，版本发行号中的每一部分对应一个字节。例如，2.6.10 对应的 LINUX_VERSION_CODE 是 132618（即 0x02060a）（注 2）。使用这个宏，我们很容易确定正在使用的内核版本。

KERNEL_VERSION(major,minor,release)

宏 KERNEL_VERSION 以组成版本号的三部分（三个整数）为参数，创建整数的版本号。例如，KERNEL_VERSION(2,6,10) 扩展为 132618。这个宏在我们需要将当前版本和一个已知的检查点比较时非常有用。

通过检查 KERNEL_VERSION 和 LINUX_VERSION_CODE 而使用预处理条件，能够解决大部分基于内核版本的依赖性问题。然而，我们不应该胡乱使用 #ifdef 条件语句将整个驱动程序代码弄得杂乱无章。最好的一个解决方法就是将所有相关的预处理条件语句集中存放在一个特定的头文件里。一般而言，依赖于特定版本（或平台）的代码应该隐藏在低层宏或者函数之后。之后，高层代码可直接调用这些函数，而无需关注低层细节。用这种方式编写的代码便于阅读，同时更为健壮。

平台依赖

每种计算机平台都有自己的独特特性，内核设计者可以充分利用这些特性来达到目标平台上目标文件的最优性能。

对于应用程序开发人员，他们必须将程序代码和预编译过的库链接并且遵循参数传递规则。而内核开发人员则不同，他们可以根据不同需求将某些寄存器指定为特定用途 ——

注 2： 这允许在稳定版本之间可存在 256 个开发版本。

实际上他们也的确这么做了。而且内核代码可以针对某个CPU家族的某种特定处理器进行优化，从而充分利用目标平台的特性。和应用程序以二进制形式的发布不同，内核需要发布源码，针对目标平台定制编译后才能达到对某个特定计算机集合的优化。

例如，IA32（x86）架构可划分为几个不同的处理器类型。老的80386处理器至今仍然被支持着，尽管从现代标准来讲，它的指令集相对受限。这种架构上更为现代的处理器已经引入了大量新的能力，包括进入内核的更快指令、进程间锁定指令、数据复制指令等等。更新的处理器（使用正确的模式）还能够处理36位（或者更大）的物理地址，从而允许处理器寻址高于4GB的物理内存。其他处理器家族也存在类似的增强。根据不同的配置选现，内核可以使用这些附加的功能。

显然，如果模块和某个给定内核工作，它也必须和内核一样了解目标处理器。这样，*vermagic.o*可再次帮助我们。在装载模块时，内核会检查处理器相关的配置选项以便确保模块匹配于运行中的内核。如果模块在不同选项下编译，则不会装载该模块。

如果读者打算编写一个驱动程序用于一般性的发布，则最好考虑好如何支持可能的不同处理器变种。当然，最好的办法是用GPL兼容许可证来发布自己的驱动程序，并将其贡献给内核主分支。如果不打算这么做，以源代码形式及一组用于编译的脚本发布自己的驱动程序则是最好的办法。许多供应商已发布了一些工具使得这个工作变得更加容易。如果读者必须以二进制方式发布自己的驱动程序，则需要检查目标发行版提供的不同内核，并为每个内核提供模块的一个版本。也请关注自发行版生产以来厂商已发布的任何勘误内核。当然，正如第一章“许可证条款”中所讨论的，许可证问题也需要考虑到。作为常规，以源代码形式发布自己的作品是最容易被其他人接受的方式。

内核符号表

在上面的讨论中，我们了解到*insmod*使用公共内核符号表来解析模块中未定义的符号。公共内核符号表中包含了所有的全局内核项（即函数和变量）的地址，这是实现模块化驱动程序所必需的。当模块被装入内核后，它所导出的任何符号都会变成内核符号表的一部分。在通常情况下，模块只需实现自己的功能，而无需导出任何符号。但是，如果其他模块需要从某个模块中获得好处时，我们也可以导出符号。

新模块可以使用由我们自己的模块导出的符号，这样，我们可以在其他模块上层叠新的模块。模块层叠技术也使用在很多主流的内核源代码中。例如，*msdos*文件系统依赖于由*fat*模块导出的符号；而每个USB输入设备模块层叠在*usbcore*和*input*模块之上。

模块层叠技术在复杂的项目中非常有用。如果以设备驱动程序的形式实现一个新的软件抽象，则可以为硬件相关的实现提供一个“插头”。例如，video-for-linux驱动程序组划

分出了一个通用模块，它导出的符号可供下层与具体硬件相关的驱动程序使用。根据所安装的硬件的不同，我们加载通用的video模块以及与具体硬件相关的特定模块。另外，并口支持以及大量可插拔设备的处理（比如USB内核子系统）都使用了类似的层叠方法。图2-2中给出了并口子系统中的层叠方式，箭头显示了模块之间以及和内核编程接口之间的通信情况。

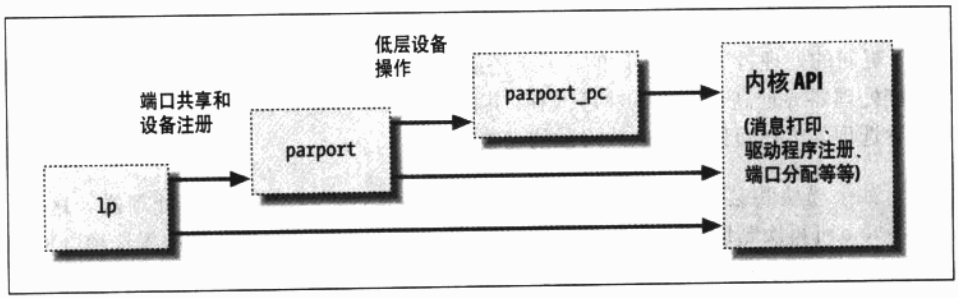


图2-2：并口驱动程序模块的层叠

*modprobe*是处理层叠模块的一个实用工具。它的功能在很大程度上和*insmod*类似，但是它除了装入指定模块外还同时装入指定模块所依赖的其他模块。因此，一个*modprobe*命令有时候相当于调用几次*insmod*命令（然而，在从当前目录装入自己的模块时仍然需要使用*insmod*，因为*modprobe*只能从标准的已安装模块目录中搜索需要装入的模块）。

通过上述层叠技术，我们可以将模块划分为多个层，通过简化每个层可缩短开发时间。这种方法和我们在第一章中提到的机制和策略的分离有点类似。

Linux内核头文件提供了一个方便的方法来管理符号对模块外部的可见性，从而减少了可能造成的名字空间污染（名字空间中的名称可能会和内核其他地方定义的名称发生冲突），并且适当隐藏信息。如果一个模块需要向其他模块导出符号，则应该使用下面的宏。

```
EXPORT_SYMBOL(name);
EXPORT_SYMBOL_GPL(name);
```

这两个宏均用于将给定的符号导出到模块外部。_GPL版本使得要导出的模块只能被GPL许可证下的模块使用。符号必须在模块文件的全局部分导出，不能在函数中导出，这是因为上面这两个宏将被扩展为一个特殊变量的声明，而该变量必须是全局的。该变量将在模块可执行文件的特殊部分（即一个“ELF段”）中保存，在装载时，内核通过这个段来寻找模块导出的变量（感兴趣的读者可以查阅*<linux/module.h>*获得更详细的信息）。

预备知识

我们离真正的模块代码越来越近了。但是，我们首先还需要了解其他一些需要在模块源代码文件中出现的东西。内核是一个特定的环境，对需要和它接口的代码有其自己的一些要求。

大部分内核代码中都要包含相当数量的头文件，以便获得函数、数据类型和变量的定义。我们将在用到这些文件时向读者介绍，但有几个头文件是专门用于模块的，因此必须出现在每个可装载的模块中。故而，所有的模块代码中都包含下面两行代码：

```
#include <linux/module.h>
#include <linux/init.h>
```

module.h 包含有可装载模块需要的大量符号和函数的定义。包含 *init.h* 的目的是指定初始化和清除函数，就像我们在“hello world”示例模块中看到的那样，下面的小节中我们还将再次看到。大部分模块还包括 *moduleparam.h* 头文件，这样我们就可以在装载模块时向模块传递参数；我们马上就可以看到如何具体使用。

尽管不是严格要求的，但模块应该指定代码所使用的许可证。为此，我们只需要包含 MODULE_LICENSE 行：

```
MODULE_LICENSE("GPL");
```

内核能够识别的许可证有“GPL”（任一版本的GNU通用公共许可证）、“GPL v2”（GPL版本2）、“GPL and additional rights（GPL及附加权利）”、“Dual BSD/GPL（BSD/GPL双重许可证）”、“Dual MPL/GPL（MPL/GPL双重许可证）”以及“Proprietary（专有）”。如果一个模块没有显式地标记为上述内核可识别的许可证，则会被假定是专有的，而内核装载这种模块就会被“污染”。如同我们在第一章“许可证条款”中提到的，内核开发者不太愿意帮助因为装载专有模块而遇到问题的用户。

可在模块中包含的其他描述性定义包括 MODULE_AUTHOR（描述模块作者）、MODULE_DESCRIPTION（用来说明模块用途的简短描述）、MODULE_VERSION（代码修订号；有关版本字符串的创建惯例，请参考 *<linux/module.h>* 中的注释）、MODULE_ALIAS（模块的别名）以及 MODULE_DEVICE_TABLE（用来告诉用户空间模块所支持的设备）。我们将在第十一章讨论 MODULE_ALIAS，并在第十二章讨论 MODULE_DEVICE_TABLE。

上述MODULE_声明可出现在源文件中源代码函数以外的任何地方。但新近的内核编码习惯是将这些声明放在文件的最后。

初始化和关闭

前面已经提到，模块的初始化函数负责注册模块所提供的任何设施。这里的设施指的是一个可以被应用程序访问的新功能，它可能是一个完整的驱动程序或者仅仅是一个新的软件抽象。初始化函数的实际定义通常如下所示：

```
static int __init initialization_function(void)
{
    /* 这里是初始化代码 */
}
module_init(initialization_function);
```

初始化函数应该被声明为static，因为这种函数在特定文件之外没有其他意义。因为一个模块函数如果要对内核其他部分可见，则必须被显式导出，因此这并不是什么强制性规则。上述定义中的__init标记看起来似乎有点陌生，它对内核来讲是一种暗示，表明该函数仅在初始化期间使用。在模块被装载之后，模块装载器就会将初始化函数扔掉，这样可将该函数占用的内存释放出来，以作他用。__init和__initdata的使用是可选的，虽然有点繁琐，但是很值得使用。注意，不要在结束初始化之后仍要使用的函数(或者数据结构)上使用这两个标记。在内核源代码中可能还会遇到__devinit和__devinitdata，只有在内核未被配置为支持热插拔设备的情况下，这两个标记才会被翻译为__init和__initdata。我们将在第十四章讲述热插拔支持。

*module_init*的使用是强制性的。这个宏会在模块的目标代码中增加一个特殊的段，用于说明内核初始化函数所在的位置。没有这个定义，初始化函数永远不会被调用。

模块可以注册许多不同类型的设施，包括不同类型的设备、文件系统、密码变换等。对每种设施，对应有具体的内核函数用来完成注册。传递到内核注册函数中的参数通常是指向用来描述新设施及设施名称的数据结构指针，而数据结构通常包含指向模块函数的指针，这样，模块体中的函数就会在恰当的时间被内核调用。

能够注册的设施类型超出了在第一章中给出的设备类型列表，它们包括串口、杂项设备、sysfs入口、*/proc*文件、可执行域以及线路规程（line discipline）等。很多可注册的设施所支持的功能属于“软件抽象”范畴，而不与任何硬件直接相关。这种类型的设施能够被注册，是因为它们能够以某种方式集成到驱动程序功能当中（如*/proc*文件系统以及线路规程）。

还有其他一些设施可以注册为特定驱动程序的附加功能，但是它们的用途有限，因而不在这里具体讨论；它们使用前面“内核符号表”一节中提到的层叠技术。如果读者想进一步了解，可以在内核源文件中grep EXPORT_SYMBOL，并找出由不同驱动程序提供的

入口点。另外，大部分注册函数名字带有register_前缀，因此找到它们的另一种方法是在内核源码中grep register_。

清除函数

每个重要的模块都需要一个清除函数，该函数在模块被移除前注销接口并向系统中返回所有资源。该函数定义如下：

```
static void _ _exit cleanup_function(void)
{
    /* 这里是清除代码 */
}

module_exit(cleanup_function);
```

清除函数没有返回值，因此被声明为void。__exit修饰词标记该代码仅用于模块卸载（编译器将把该函数放在特殊的ELF段中）。如果模块被直接内嵌到内核中，或者内核的配置不允许卸载模块，则被标记为__exit的函数将被简单地丢弃。出于以上原因，被标记为__exit的函数只能在模块被卸载或者系统关闭时被调用，其他的任何用法都是错误的。和前面类似，*module_exit*声明对于帮助内核找到模块的清除函数是必需的。

如果一个模块未定义清除函数，则内核不允许卸载该模块。

初始化过程中的错误处理

当我们在内核中注册设施时，要时刻铭记注册可能会失败。即使是最简单的动作，都需要内存分配，而所需要的内存可能无法获得。因此模块代码必须始终检查返回值，并确保所请求的操作已真正成功。

如果在注册设施时遇到任何错误，首先要判断模块是否可以继续初始化。通常，在某个注册失败后可以通过降低功能来继续运转。因此，只要可能，模块应该继续向前并尽可能提供其功能。

如果在发生了某个特定类型的错误之后无法继续装载模块，则要将出错之前的任何注册工作撤销掉。Linux中没有记录每个模块都注册了哪些设施，因此，当模块的初始化出现错误之后，模块必须自行撤销已注册的设施。如果由于某种原因我们未能撤销已注册的设施，则内核会处于一种不稳定状态，这是因为内核中包含了一些指向并不存在的代码的内部指针。在这种情况下，唯一有效的解决办法是重新引导系统。因此，我们必须在初始化过程出现错误时认真完成正确的工作。

错误恢复的处理有时使用goto语句比较有效。通常情况下我们很少使用goto，但在处理错误时（可能是唯一的情况）它却非常有用。错误情况下的goto的仔细使用可避免大量复杂的、高度缩进的“结构化”逻辑。因此，内核经常使用goto来处理错误。

不管初始化过程在什么时刻失败，下面的例子（使用了虚构的注册和撤销注册函数）都能正确工作：

```
int _ _init my_init_function(void)
{
    int err;

    /* 使用指针和名称注册 */
    err = register_this(ptr1, "skull");
    if (err) goto fail_this;
    err = register_that(ptr2, "skull");
    if (err) goto fail_that;
    err = register_those(ptr3, "skull");
    if (err) goto fail_those;

    return 0; /* 成功 */

  fail_those: unregister_that(ptr2, "skull");
  fail_that: unregister_this(ptr1, "skull");
  fail_this: return err; /* 返回错误 */
 }
```

这段代码准备注册三个（虚构的）设施。在出错的时候使用goto语句，它将只撤销出错时刻以前所成功注册的那些设施。

另一种观点不支持goto的使用，而是记录任何成功注册的设施，然后在出错的时候调用模块的清除函数。清除函数将仅仅回滚已成功完成的步骤。然而，这种替代方法需要更多的代码和CPU时间，因此在追求效率的代码中使用goto语句仍然是最好的错误恢复机制。

*my_init_module*的返回值err是一个错误编码。在Linux内核中，错误编码是定义在*<linux/errno.h>*中的负整数。如果我们不想使用其他函数返回的错误编码，而想使用自己的错误编码，则应该包含*<linux/errno.h>*，以使用诸如-ENODEV、-ENOMEM之类的符号值。每次返回合适的错误编码是一个好习惯，因为用户程序可以通过*perror*函数或类似的途径将它们转换为有意义的字符串。

显然，模块的清除函数需要撤销初始化函数所注册的所有设施，并且习惯上（但不是必须的）以相反于注册的顺序撤销设施：

```
void _ _exit my_cleanup_function(void)
{
```

```
    unregister_those(ptr3, "skull");
    unregister_that(ptr2, "skull");
    unregister_this(ptr1, "skull");
    return;
}
```

如果初始化和清除工作涉及很多设施，则goto方法可能变得难以管理，因为所有用于清除设施的代码在初始化函数中重复，同时一些标号交织在一起。因此，有时候我们需要考虑重新构思代码的结构。

每当发生错误时从初始化函数中调用清除函数，这种方法将减少代码的重复并且使代码更清晰、更有条理。当然，清除函数必须在撤销每项设施的注册之前检查它的状态。下面是这种方法的简单示例：

```
struct something *item1;
struct somethingelse *item2;
int stuff_ok;

void my_cleanup(void)
{
    if (item1)
        release_thing(item1);
    if (item2)
        release_thing2(item2);
    if (stuff_ok)
        unregister_stuff();
    return;
 }

int _ _init my_init(void)
{
    int err = -ENOMEM;

    item1 = allocate_thing(arguments);
    item2 = allocate_thing2(arguments2);
    if (!item1 || !item2)
        goto fail;
    err = register_stuff(item1, item2);
    if (!err)
        stuff_ok = 1;
    else
        goto fail;
    return 0; /* 成功 */

  fail:
    my_cleanup();
    return err;
}
```

如这段代码所示，根据调用的注册/分配函数的语义，我们可以使用或不使用外部标志来标记每个初始化步骤的成功。不管是否需要使用标志，这种方式的初始化能够很好地

扩展到对大量设施的支持，因此比前面介绍的技术更具优越性。然而需要注意的是，因为清除函数被非退出代码调用，因此不能将清除函数标记为__exit。

模块装载竞争

目前为止，我们已经简单讨论了模块装载中的一个重要方面：竞态（race condition）。如果在编写初始化函数时不够仔细，就可能危及整个系统的稳定性。在本书后面的章节中将进一步讨论竞态问题，本节将阐述一些要点。

首先要始终铭记的是，在注册完成之后，内核的某些部分可能会立即使用我们刚刚注册的任何设施。换句话说，在初始化函数还在运行的时候，内核就完全可能会调用我们的模块。因此，在首次注册完成之后，代码就应该准备好被内核的其他部分调用；在用来支持某个设施的所有内部初始化完成之前，不要注册任何设施。

我们还必须考虑，当初始化失败而内核的某些部分已经使用了模块所注册的某个设施时应该如何处理。如果这种情况可能发生在我们的模块上，则根本不应该出现初始化失败的情况，毕竟模块已经成功导出了可用的功能及符号。如果初始化一定要失败，则应该仔细处理内核其他部分正在进行的操作，并且要等待这些操作的完成。

模块参数

由于系统的不同，驱动程序需要的参数也许会发生变化。这包括设备编号（下一章讨论）以及其他一些用来控制驱动程序操作方式的参数。例如，SCSI适配器的驱动程序经常要处理一些选项，这些选项用来控制标记命令队列的使用，而集成设备电路（Integrated Device Electronics，IDE）驱动程序允许用户控制DAM操作。如果读者的驱动程序用来控制一些早期的硬件，也许需要明确告知驱动程序硬件的I/O端口或者I/O内存地址的位置。为满足这种需求，内核允许对驱动程序指定参数，而这些参数可在装载驱动程序模块时改变。

这些参数的值可在运行*insmod*或*modprobe*命令装载模块时赋值，而*modprob*还可以从它的配置文件（*/etc/modprob.conf*）中读取参数值。这两个命令可在命令行接受几种参数类型的赋值。为了演示这种功能，我们假定对本章前面的“hello world”模块（命名为*hellop*）做了一些必要的增强。我们添加了两个参数：一个是整数值，其名称为howmany；另一个是字符串，名称为whom。在装载这个增强的模块时，将向whom问候howmany次。这样，我们可用下面的命令行来装载该模块：

```
insmod hellop howmany=10 whom="Mom"
```

上面这条命令的效果会让 *hellop* 打印 10 次 “hello, Mom”。

当然，在 *insmod* 改变模块参数之前，模块必须让这些参数对 *insmod* 命令可见。参数必须使用module_param宏来声明，这个宏在 *moduleparam.h* 中定义。module_param需要三个参数：变量的名称、类型以及用于 sysfs 入口项的访问许可掩码。这个宏必须放在任何函数之外，通常是在源文件的头部。这样，*hellop* 通过下面的代码来声明它的参数并使之对 *insmod* 可见：

```
static char *whom = "world";
static int howmany = 1;
module_param(howmany, int, S_IRUGO);
module_param(whom, charp, S_IRUGO);
```

内核支持的模块参数类型如下：

bool

invbool

布尔值（取 true 或 false），关联的变量应该是 int 型。invbool 类型反转其值，也就是说，true 值变成 false，而 false 变成 true。

charp

字符指针值。内核会为用户提供的字符串分配内存，并相应设置指针。

int

long

short

uint

ulong

ushort

具有不同长度的基本整数值。以 u 开头的类型用于无符号值。

模块装载器也支持数组参数，在提供数组值时用逗号划分各数组成员。要声明数组参数，需要使用下面的宏：

```
module_param_array(name,type,num,perm);
```

其中，name 是数组的名称（也就是参数的名称），type 是数组元素的类型，num 是一个整数变量，而perm是常见的访问许可值。如果在装载时设置数组参数，则num会被设置为用户提供的值的个数。模块装载器会拒绝接受超过数组大小的值。

如果我们需要的类型不在上面所列出的清单中，模块代码中的钩子可让我们来定义这些类型。具体的细节请参阅 *moduleparam.h* 文件。所有的模块参数都应该给定一个默认值；

*insmod*只会在用户明确设置了参数的值的情况下才会改变参数的值。模块可以根据默认值来判断是否是一个显式指定的参数。

*module_param*中的最后一个成员是访问许可值，我们应使用*<linux/stat.h>*中存在的定义。这个值用来控制谁能够访问sysfs中对模块参数的表述。如果perm被设置为0，就不会有对应的sysfs入口项；否则，模块参数会在*/sys/module*（注3）中出现，并设置为给定的访问许可。如果对参数使用S_IRUGO，则任何人均可读取该参数，但不能修改；S_IRUGO|S_IWUSR允许root用户修改该参数。注意，如果一个参数通过sysfs而被修改，则如同模块修改了这个参数的值一样，但是内核不会以任何方式通知模块。大多数情况下，我们不应该让模块参数是可写的，除非我们打算检测这种修改并作出相应的动作。

在用户空间编写驱动程序

首次接触内核的Unix程序员可能对编写模块比较紧张，然而编写用户空间程序来直接对设备端口进行读写就容易多了。

相对于内核空间编程，用户空间编程具有自己的一些优点。有时候编写一个所谓的用户空间驱动程序是替代内核空间驱动程序的一个不错的方法。在这一小节，我们将讨论编写用户空间驱动程序的几个理由。但本书主要讲述内核空间的驱动程序，因此除了这里的讨论之外，我们不会进一步深入讨论这个话题。

用户空间驱动程序的优点可以归纳如下：

- 可以和整个C库链接。驱动程序不用借助外部程序（即前面提到的和驱动程序一起发行的用于提供策略的用户程序）就可以完成许多非常规任务。
- 可以使用通常的调试器调试驱动程序代码，而不用费力地调试正在运行的内核。
- 如果用户空间驱动程序被挂起，则简单地杀掉它就行了。驱动程序带来的问题不会挂起整个系统，除非所驱动的硬件已经发生严重故障。
- 和内核内存不同，用户内存可以换出。如果驱动程序很大但是不经常使用，则除了正在使用的情况之外，不会占用太多内存。
- 良好设计的驱动程序仍然支持对设备的并发访问。
- 如果读者必须编写封闭源码的驱动程序，则用户空间驱动程序可更加容易地避免因为修改内核接口而导致的不明确的许可问题。

注3： 但在本书编写时，内核开发者正在讨论是否要将参数转移到sysfs的其他地方。

例如，USB驱动程序可在用户空间编写；具体可参阅libusb项目（libusb.sourceforge.net，该项目还比较“年轻”），以及内核源代码中的“gadgetfs”。X服务器是用户空间驱动程序的另一个例子。它十分清楚硬件可以做什么、不可以做什么，并且为所有的X客户提供图形资源。然而，值得注意的是目前基于帧缓冲区（frame-buffer）的图形环境正在慢慢成为发展趋势。这种环境下对于实际的图形操作，X服务器仅仅是一个基于真正内核空间驱动程序的服务器。

通常，用户空间的驱动程序被实现为一个服务器进程，其任务是替代内核作为硬件控制的唯一代理。客户应用程序可连接到该服务器并和设备执行实际的通信；这样，好的驱动程序进程可允许对设备的并发访问。其实这就是X服务器的本质。

除了具备上述优点外，用户空间驱动程序也有很多缺点，下面列出其中最重要的几点：

- 中断在用户空间中不可用。对该限制，在某些平台上也有相应的解决办法，比如IA32架构上的*vm86*系统调用。
- 只有通过*mmap*映射*/dev/mem*才能直接访问内存，但只有特权用户才可以执行这个操作。
- 只有在调用*ioperm*或*iopl*后才可以访问I/O端口。然而并不是所有平台都支持这两个系统调用，并且访问*/dev/port*可能非常慢，因而并非十分有效。同样只有特权用户才能引用这些系统调用和访问设备文件。
- 响应时间很慢。这是因为在客户端和硬件之间传递数据和动作需要上下文切换。
- 更严重的是，如果驱动程序被换出到磁盘，响应时间将令人难以忍受。使用*mlock*系统调用或许可以缓解这一问题，但由于用户空间程序一般需要链接多个库，因此通常需要占用多个内存页。同样，*mlock*也只有特权用户才能引用。
- 用户空间中不能处理一些非常重要的设备，包括（但不限于）网络接口和块设备等。

如上所述，我们看到用户空间驱动程序毕竟做不了太多的工作。然而依然存在一些有意义的应用，例如对SCSI扫描设备（由包SANE实现）和CD刻录设备（由*cdrecord*和其他工具实现）的支持。这两种情况下，用户空间驱动程序都依赖内核空间驱动程序“SCSI generic”，它导出底层通用的SCSI功能到用户空间程序，然后再由用户空间驱动程序驱动自己的硬件。

有一种情况适合在用户空间处理，这就是当我们准备处理一种新的、不常见的硬件时。在用户空间中我们可以研究如何管理这个硬件而不用担心挂起整个系统。一旦完成，就很容易将户空间驱动程序封装到内核模块中。

快速参考

本节将总结本章中提到的内核函数、变量、宏以及/proc文件，可以作为对这些内容的一个参考。每一项都会在相关头文件之后列出。从本章开始，以后每一章里都会有类似的一节来总结引入的新符号。本节中出现的条目会以它们在文中出现的顺序列出：

insmod
modprobe
rmmod

用来装载模块到正运行的内核和移除模块的用户空间工具。

```
#include <linux/init.h>
module_init(init_function);
module_exit(cleanup_function);
```

用于指定模块的初始化和清除函数的宏。

```
__init
__initdata
__exit
__exitdata
```

仅用于模块初始化或清除阶段的函数(__init和__exit)和数据(__initdata和__exitdata)标记。标记为初始化的项目会在初始化结束后丢弃；而退出项目在内核未被配置为可卸载模块的情况下被丢弃。内核通过将相应的目标对象放置在可执行文件的特殊ELF段中而让这些标记起作用。

```
#include <linux/sched.h>
```

最重要的头文件之一。该文件包含驱动程序使用的大部分内核API的定义，包括睡眠函数以及各种变量声明。

```
struct task_struct *current;
```

当前进程。

```
current->pid
current->comm
```

当前进程的进程ID和命令名。

```
obj-m
```

由内核构造系统使用的makefile符号，用来确定在当前目录中应构造哪些模块。

/sys/module
/proc/modules

*/sys/module*是sysfs目录层次结构中包含当前已装载模块信息的目录。*/proc/*

modules 是早期用法，只在单个文件中包括这些信息，其中包含了模块名称、每个模块使用的内存总量以及使用计数等。每一行之后还追加有额外的字符串，用来指定模块的当前活动标志。

vermagic.o

内核源代码目录中的一个目标文件，它描述了模块的构造环境。

```
#include <linux/module.h>
```

必需的头文件，它必须包含在模块源代码中。

```
#include <linux/version.h>
```

包含所构造内核版本信息的头文件。

```
LINUX_VERSION_CODE
```

整数宏，在处理版本依赖的预处理条件语句中非常有用。

```
EXPORT_SYMBOL (symbol);
EXPORT_SYMBOL_GPL (symbol);
```

用来导出单个符号到内核的宏。第二个宏将导出符号的使用限于GPL许可证下的模块。

```
MODULE_AUTHOR(author);
MODULE_DESCRIPTION(description);
MODULE_VERSION(version_string);
MODULE_DEVICE_TABLE(table_info);
MODULE_ALIAS(alternate_name);
```

在目标文件中添加关于模块的文档信息。

```
module_init(init_function);
module_exit(exit_function);
```

用来声明模块初始化和清除函数的宏。

```
#include <linux/moduleparam.h>
module_param(variable, type, perm);
```

用来创建模块参数的宏，用户可在装载模块时（或者对内建代码引导时）调整这些参数的值。其中的类型可以是bool、charp、int、invbool、long、short、ushort、uint、ulong或者intarray。

```
#include <linux/kernel.h>
int printk(const char * fmt, ...);
```

函数 *printf* 的内核代码。

第三章

字符设备驱动程序



本章的目标是编写一个完整的字符设备驱动程序。我们开发字符设备驱动程序的原因是因为此类驱动程序适合于大多数简单的硬件设备，而且比起块设备或网络驱动程序（我们将在后面的章节中讲述这两类驱动程序）更加易于理解。我们的最终目标是编写一个模块化的字符设备驱动程序，但本章不会讨论模块化的相关问题。

贯穿全章我们将介绍一些代码段，它们取自一个真正的设备驱动程序：*scull*，即“Simple Character Utility for Loading Localities，区域装载的简单字符工具”的缩写。*scull*是一个操作内存区域的字符设备驱动程序，这片内存区域就相当于一个设备。本章中，因为*scull*的特殊之处，“设备”这个词可与“scull所使用的内存区域”互换使用。

*scull*的优点在于它不和硬件相关，而只是操作从内核中分配的一些内存。任何人都可以编译和运行*scull*，而且还可以将*scull*移植到Linux支持的所有计算机平台上。但另一方面，除了展示内核和字符设备驱动程序之间的接口并且让用户运行某些测试例程外，*scull*设备做不了任何“有用的”事情。

scull的设计

编写驱动程序的第一步就是定义驱动程序为用户程序提供的能力（机制）。由于我们的“设备”是计算机内存的一部分，所以可以利用它随意地做我们想做的事情。它可以是顺序或随机存取设备，也可以是一个或多个设备等。

为了让*scull*能够为编写真正的设备驱动程序提供一个样板，我们将展示怎样在计算机内存之上实现若干设备抽象，而且每个都具有自己的特点。

*scull*的源代码实现了下列设备，我们将由模块实现的每种设备称作一种“类型”：

scull0 ~ scull3

这四个设备分别由一个全局且持久的内存区域组成。“全局”是指：如果设备被多次打开，则打开它的所有文件描述符可共享该设备所包含的数据。“持久”是指：如果设备关闭后再打开，则其中的数据不会丢失。可以使用常用命令来访问和测试这个设备，如 *cp*、*cat* 以及 shell 的 I/O 重定向等。

scullpipe0 ~ scullpipe3

这四个 FIFO（先入先出）设备与管道类似。一个进程读取由另一个进程写入的数据。如果多个进程读取同一个设备，它们就会为数据发生竞争。*scullpipe* 的内部实现将说明在不借助于中断的情况下如何实现阻塞式和非阻塞式读/写操作。虽然实际的驱动程序使用硬件中断与它们的设备保持同步，但阻塞式和非阻塞式操作是一个重要内容，并且有别于中断处理（第十章将作介绍）。

scullsingle

scullpriv

sculluid

scullwuid

这些设备与 *scull0* 相似，但在何时允许 *open* 操作方面有一些限制。第一个（*scullsingle*）一次只允许一个进程使用该驱动程序，而 *scullpriv* 对每个虚拟控制台（或X终端会话）是私有的，这是因为每个控制台/终端上的进程将获取不同的内存区。*sculluid* 和 *scullwuid* 可被多次打开，但每次只能由一个用户打开；如果另一个用户锁定了该设备，*sculluid* 将返回“Device Busy”的错误，而 *scullwuid* 则实现了阻塞式 *open*。这些 *scull* 设备的变种混淆了“机制”和“策略”，但这类处理是值得去了解的，因为某些真正的设备需要类似的管理方式。

每个 *scull* 设备都展示了驱动程序的不同功能，也提出了不同的难点。本章主要涉及 *scull0 ~ scull3* 的内部结构；更为复杂的设备将在第六章介绍：*scullpipe* 在“阻塞式 I/O 示例”一节中讲述，而其他设备在“设备文件的访问控制”中介绍。

主设备号和次设备号

对字符设备的访问是通过文件系统内的设备名称进行的。那些名称被称为特殊文件、设备文件，或者简单称之为文件系统树的节点，它们通常位于 */dev* 目录。字符设备驱动程序的设备文件可通过 *ls -l* 命令输出的第一列中的“c”来识别。块设备也出现在 */dev* 下，但它们由字符“b”标识。本章主要关注字符设备，不过下面介绍的许多内容也同样适用于块设备。

如果执行 *ls -l* 命令，则可在设备文件项的最后修改日期前看到两个数（用逗号分隔），这个位置通常显示的是文件的长度；而对设备文件，这两个数就是相应设备的主设备号和次设备号。下面的列表给出了典型系统中的一些设备。它们的主设备号是1、4、7和10，而次设备号是1、3、5、64、65和129。

```
crw-rw-rw-   1 root     root      1,   3 Apr 11  2002 null
crw-------   1 root     root     10,   1 Apr 11  2002 psaux
crw-------   1 root     root      4,   1 Oct 28 03:04 tty1
crw-rw-rw-   1 root     tty       4,  64 Apr 11  2002 ttys0
crw-rw----   1 root     uucp      4,  65 Apr 11  2002 ttyS1
crw--w----   1 vcsa     tty       7,   1 Apr 11  2002 vcs1
crw--w----   1 vcsa     tty       7, 129 Apr 11  2002 vcsa1
crw-rw-rw-   1 root     root      1,   5 Apr 11  2002 zero
```

通常而言，主设备号标识设备对应的驱动程序。例如，*/dev/null*和*/dev/zero*由驱动程序1管理，而虚拟控制台和串口终端由驱动程序4管理；类似地，*vcs1*和*vcsa1*设备都由驱动程序7管理。现代的Linux内核允许多个驱动程序共享主设备号，但我们看到的大多数设备仍然按照“一个主设备号对应一个驱动程序”的原则组织。

次设备号由内核使用，用于正确确定设备文件所指的设备。依赖于驱动程序的编写方式（将在下面阐述），我们可以通过次设备号获得一个指向内核设备的直接指针，也可将次设备号当作设备本地数组的索引。不管用哪种方式，除了知道次设备号用来指向驱动程序所实现的设备之外，内核本身基本上不关心关于次设备号的任何其他信息。

设备编号的内部表达

在内核中，dev_t类型（在*<linux/types.h>*中定义）用来保存设备编号——包括主设备号和次设备号。在内核的2.6.0版本中，dev_t是一个32位的数，其中的12位用来表示主设备号，而其余20位用来表示次设备号。当然，我们的代码不应该对设备编号的组织做任何假定，而应该始终使用*<linux/kdev_t.h>*中定义的宏。比如，要获得dev_t的主设备号或次设备号，应使用：

```
MAJOR(dev_t dev);
MINOR(dev_t dev);
```

相反，如果需要将主设备号和次设备号转换成dev_t类型，则使用：

```
MKDEV(int major, int minor);
```

注意，2.6内核可以容纳大量的设备，而先前的内核版本却限于255个主设备号和255个次设备号。我们可以认为更宽的范围在相当长的时间内是足够的，然而，计算领域中充

斥着大量的类似假定。因此，我们会看到dev_t格式在将来会发生变化；但如果我们小心地编写自己的驱动程序，那么这种变化不会带来问题。

分配和释放设备编号

在建立一个字符设备之前，我们的驱动程序首先要做的事情就是获得一个或者多个设备编号。完成该工作的必要函数是*register_chrdev_region*，该函数在*<linux/fs.h>*中声明：

```
int register_chrdev_region(dev_t first, unsigned int count,
                           char *name);
```

其中，first是要分配的设备编号范围的起始值。first的次设备号经常被置为0，但对该函数来讲并不是必需的。count是所请求的连续设备编号的个数。注意，如果count非常大，则所请求的范围可能和下一个主设备号重叠，但只要我们所请求的编号范围是可用的，则不会带来任何问题。最后，name是和该编号范围关联的设备名称，它将出现在*/proc/devices*和sysfs中。

和大部分内核函数一样，*register_chrdev_region*的返回值在分配成功时为0。在错误情况下，将返回一个负的错误码，并且不能使用所请求的编号区域。

如果我们提前明确知道所需要的设备编号，则*register_chrdev_region*会工作得很好。但是，我们经常不知道设备将要使用哪些主设备号；因此，Linux内核开发社区一直在努力转向设备编号的动态分配。在运行过程中使用下面的函数，内核将会为我们恰当分配所需要的主设备号：

```
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor,
                        unsigned int count, char *name);
```

在上面这个函数中，dev是仅用于输出的参数，在成功完成调用后将保存已分配范围的第一个编号。firstminor应该是要使用的被请求的第一个次设备号，它通常是0。count和name参数与*register_chrdev_region*函数是一样的。

不论采用哪种方法分配设备编号，都应该在不再使用它们时释放这些设备编号。设备编号的释放需要使用下面的函数：

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

通常，我们在模块的清除函数中调用*unregister_chrdev_region*函数。

上面的函数为驱动程序的使用分配设备编号，但是它们并没有告诉内核关于拿来这些编号要做什么工作。在用户空间程序可访问上述设备编号之前，驱动程序需要将设备编号

和内部函数连接起来，这些内部函数用来实现设备的操作。我们会马上讨论如何实现这种连接，但在此之前还需要进一步讨论有关设备号的内容。

动态分配主设备号

一部分主设备号已经静态地分配给了大部分常见设备。在内核源码树的*Documentation/devices.txt*文件中可以找到这些设备的清单。将某个已经分配好的静态编号用于新驱动程序的机会非常小，但是尚未被分配的新编号是可用的。因此，作为驱动程序作者，我们有如下选择：可以简单选定一个尚未被使用的编号，或者通过动态方式分配主设备号。如果使用驱动程序的人只有我们自己，则选定一个编号的方法永远行得通；然而，一旦驱动程序被广泛使用，随机选定的主设备号可能造成冲突和麻烦。

因此，对于一个新的驱动程序，我们强烈建议读者不要随便选择一个当前未使用的设备号作为主设备号，而应该使用动态分配机制获取主设备号。换句话说，驱动程序应该始终使用*alloc_chrdev_region*而不是*register_chrdev_region*函数。

动态分配的缺点是：由于分配的主设备号不能保证始终一致，所以无法预先创建设备节点。对于驱动程序的一般用法，这倒不是什么问题，因为一旦分配了设备号，就可以从*/proc/devices*中读取得到（注1）。

因此，为了加载一个使用动态主设备号的设备驱动程序，对*insmod*的调用可替换为一个简单的脚本，该脚本在调用*insmod*之后读取*/proc/devices*以获得新分配的主设备号，然后创建对应的设备文件。

典型的*/proc/devices*文件如下所示：

```
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 ttyS
  6 lp
  7 vcs
 10 misc
 13 input
 14 sound
 21 sg
180 usb
```

注1：设备信息通常能够从sysfs中更好地获得，在基于2.6内核的系统中，sysfs通常被挂载到*/sys*上。但是，让*scull*通过sysfs来导出信息的方法已超出了本书的讨论范围；我们将在第十四章讨论sysfs。

```
Block devices:
 2 fd
 8 sd
 11 sr
 65 sd
 66 sd
```

动态分配主设备号的情况下，要加载这类驱动程序模块的脚本，可以利用*awk*这类工具从*/proc/devices*中获取信息，并在*/dev*目录中创建设备文件。

下面这个名为*scull_load*的脚本是*scull*发布的一部分。使用以模块形式发行的驱动程序的用户可以在系统的*rc.local*文件中调用这个脚本，或是在需要模块时手工调用。

```
#!/bin/sh
module="scull"
device="scull"
mode="664"

# 使用传入到该脚本的所有参数调用 insmod，同时使用路径名来指定模块位置，
# 这是因为新的modutils默认不会在当前目录中查找模块。
/sbin/insmod ./$module.ko $* || exit 1

# 删除原有节点
rm -f /dev/${device}[0-3]

major=$(awk "\$2=  =\"$module\" {print \$1}" /proc/devices)

mknod /dev/${device}0 c $major 0
mknod /dev/${device}1 c $major 1
mknod /dev/${device}2 c $major 2
mknod /dev/${device}3 c $major 3

# 给定适当的组属性及许可，并修改属组。
# 并非所有的发行版都具有staff组，有些有wheel组。
group="staff"
grep -q '^staff:' /etc/group || group="wheel"

chgrp $group /dev/${device}[0-3]
chmod $mode  /dev/${device}[0-3]
```

这个脚本同样可以适用于其他驱动程序，只要重新定义变量并调整*mknod*那几行语句就可以了。该脚本创建了四个设备，因为*scull*的源码默认创建四个设备。

脚本的最后几行看起来有点奇怪：为什么要改变设备的组和访问模式呢？原因在于这个脚本必须由超级用户运行，所以新创建的设备文件自然属于root。默认的权限位只允许root对其有写访问权，而其他用户只有读权限。通常，设备节点需要不同的访问策略，因此有时需要修改访问权限。我们的脚本默认地把访问权赋于一个用户组，而读者的需求可能有所不同。第六章“设备文件的访问控制”一节中，*sculluid*的代码将会展示设备驱动程序如何实现自己的设备访问授权。

除*scull_load*之外，还有一个*scull_unload*脚本用来清除*/dev*目录下的相关设备文件并卸载这个模块。

除了使用这一对装载和卸载模块的脚本外，我们还可以编写一个init脚本，并将其保存在发行版使用的init脚本目录中（注2）。作为*scull*源码的一部分，我们提供了相当详尽和可配置的init脚本范例，名为*scull.init*。它接收约定的参数——start、stop以及restart——而且可完成*scull_load*和*scull_unload*的双重任务。

如果反复创建和删除*/dev*节点显得有些不必要的话，那么有一个解决的方法。如果只是装载和卸载单个驱动程序，则可在第一次创建设备文件之后仅使用*rmmod*和*insmod*这两个命令：因为动态设备号并不是随机生成的（注3），如果不受其他（动态）模块影响的话，可以预期获得相同的动态主设备号。在开发过程中避免脚本过长是有益的。但很明显，这个技巧不能适用于同时有多个驱动程序存在的场合。

以笔者的看法，分配主设备号的最佳方式是：默认采用动态分配，同时保留在加载甚至是编译时指定主设备号的余地。*scull*的实现采用了这种方式；它使用了一个全局变量scull_major，用来保存所选择的设备号（也有一个用于次设备号的scull_minor变量）。该变量的初始化值是SCULL_MAJOR，这个宏定义在*scull.h*中。在我们发布的源代码中，SCULL_MAJOR默认取0，即“选择动态分配”。用户可以使用这个默认值或选择某个特定的主设备号，而且既可以在编译前修改宏定义，也可以通过*insmod*命令行指定scull_major的值。最后，通过使用*scull_load*脚本，用户可以在*scull_load*的命令行中将参数传递给*insmod*（注4）。

下面是*scull.c*中用来获取主设备号的代码：

```
if (scull_major) {
    dev = MKDEV(scull_major, scull_minor);
    result = register_chrdev_region(dev, scull_nr_devs, "scull");
} else {
    result = alloc_chrdev_region(&dev, scull_minor, scull_nr_devs,
            "scull");
    scull_major = MAJOR(dev);
}
```

注2：Linux Standard要求将init脚本置于*/etc/init.d*目录下，但某些发行版仍将这些版本置于其他目录。另外，如果要在引导阶段运行某个脚本，则需要从适当的运行级别目录（例如，*.../rc3.d*）创建一个指向该脚本的链接。

注3：不过，某些内核开发人员已经预示在不久的将来将会按随机方式进行处理。

注4：init脚本*scull.init*不接受来自命令行的驱动程序选项，但它支持一个配置文件，因为这个脚本是为启动和关机时自动运行而设计的。

```
if (result < 0) {
    printk(KERN_WARNING "scull: can't get major %d\n", scull_major);
    return result;
}
```

本书中几乎所有的示例驱动程序都使用类似的代码来分配它们的主设备号。

一些重要的数据结构

读者可能会想像到，设备编号的注册仅仅是驱动程序代码必须完成的许多工作中的第一件事情而已。我们很快就会看到其他一些重要的驱动程序组件，但在此之前还需要阐述另一个话题。大部分基本的驱动程序操作涉及到三个重要的内核数据结构，分别是file_operations、file和inode。在编写真正的驱动程序之前，需要对上述结构有一个基本的认识，因此在阐述如何实现基本的驱动程序操作之前，我们将快速描述上述数据结构。

文件操作

迄今为止，我们已经为自己保留了一些设备编号，但尚未将任何驱动程序操作连接到这些编号。file_operations结构就是用来建立这种连接的。这个结构定义在*<linux/fs.h>*中，其中包含了一组函数指针。每个打开的文件（在内部由一个file结构表示，稍后就会讲到）和一组函数关联（通过包含指向一个file_operations结构的f_op字段）。这些操作主要用来实现系统调用，命名为*open*、*read*等等。我们可以认为文件是一个“对象”，而操作它的函数是“方法”，如果采用面向对象编程的术语来表达就是，对象声明的动作将作用于其本身。这是我们在Linux内核中看到的面向对象编程的第一个例证，在后面的章节中还会看到更多。

按照惯例，file_operations结构或者指向这类结构的指针称为fops（或者是与此相关的其他叫法）。这个结构中的每一个字段都必须指向驱动程序中实现特定操作的函数，对于不支持的操作，对应的字段可置为NULL值。对各个函数而言，如果对应字段被赋为NULL指针，那么内核的具体处理行为是不尽相同的，本节后面的列表会列出这些差异。

下面列出了应用程序可在某个设备上调用的所有操作。为便于查询，我们尽量使之简洁，仅仅总结了每个操作以及使用NULL时的内核默认行为。

在通读file_operations方法的清单时，我们会注意到许多参数包含有__user字符串，它其实是一种形式的文档而已，表明指针是一个用户空间地址，因此不能被直接引

用。对通常的编译来讲，__user没有任何效果，但是可由外部检查软件使用，用来寻找对用户空间地址的错误使用。

介绍完其他一些重要的数据结构后，本章其余部分将讲解最重要的一些操作并给出一些技巧、警告和实际的代码样例。由于我们尚未深入探讨内存管理、块操作和异步通知机制，其他更为复杂的操作将在以后的章节中介绍。

`struct module *owner`

第一个file_operations字段并不是一个操作；相反，它是指向“拥有”该结构的模块的指针。内核使用这个字段以避免在模块的操作正在被使用时卸载该模块。几乎在所有的情况下，该成员都会被初始化为THIS_MODULE，它是定义在*<linux/module.h>*中的一个宏。

`loff_t (*llseek) (struct file *, loff_t, int);`

方法*llseek*用来修改文件的当前读写位置，并将新位置作为（正的）返回值返回。参数loff_t一个“长偏移量”，即使在32位平台上也至少占用64位的数据宽度。出错时返回一个负的返回值。如果这个函数指针是NULL，对seek的调用将会以某种不可预期的方式修改file结构（在“file结构”一节中有描述）中的位置计数器。

`ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);`

用来从设备中读取数据。该函数指针被赋为NULL值时，将导致read系统调用出错并返回-EINVAL（“Invalid argument，非法参数”）。函数返回非负值表示成功读取的字节数（返回值为“signed size”数据类型，通常就是目标平台上的固有整数类型）。

`ssize_t (*aio_read)(struct kiocb *, char _ _user *, size_t, loff_t);`

初始化一个异步的读取操作——即在函数返回之前可能不会完成的读取操作。如果该方法为NULL，所有的操作将通过read（同步）处理。

`ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);`

向设备发送数据。如果没有这个函数，*write*系统调用会向程序返回一个-EINVAL。如果返回值非负，则表示成功写入的字节数。

`ssize_t (*aio_write)(struct kiocb *, const char __user *, size_t, loff_t *);`

初始化设备上的异步写入操作。

`int (*readdir) (struct file *, void *, filldir_t);`

对于设备文件来说，这个字段应该为NULL。它仅用于读取目录，只对文件系统有用。

```
unsigned int (*poll) (struct file *, struct poll_table_struct *);
```

*poll*方法是*poll*、*epoll*和*select*这三个系统调用的后端实现。这三个系统调用可用来查询某个或多个文件描述符上的读取或写入是否会被阻塞。*poll*方法应该返回一个位掩码，用来指出非阻塞的读取或写入是否可能，并且也会向内核提供将调用进程置于休眠状态直到I/O变为可能时的信息。如果驱动程序将*poll*方法定义为NULL，则设备会被认为既可读也可写，并且不会被阻塞。

```
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
```

系统调用*ioctl*提供了一种执行设备特定命令的方法（如格式化软盘的某个磁道，这既不是读操作也不是写操作）。另外，内核还能识别一部分*ioctl*命令，而不必调用fops表中的*ioctl*。如果设备不提供*ioctl*入口点，则对于任何内核未预先定义的请求，*ioctl*系统调用将返回错误（-ENOTTY，“No such ioctl for device，该设备无此ioctl命令”）。

```
int (*mmap) (struct file *, struct vm_area_struct *);
```

*mmap*用于请求将设备内存映射到进程地址空间。如果设备没有实现这个方法，那么*mmap*系统调用将返回-ENODEV。

```
int (*open) (struct inode *, struct file *);
```

尽管这始终是对设备文件执行的第一个操作，然而却并不要求驱动程序一定要声明一个相应的方法。如果这个入口为NULL，设备的打开操作永远成功，但系统不会通知驱动程序。

```
int (*flush) (struct file *);
```

对*flush*操作的调用发生在进程关闭设备文件描述符副本的时候，它应该执行（并等待）设备上尚为完结的操作。请不要将它同用户程序使用的*fsync*操作相混淆。目前，*flush*仅仅用于少数几个驱动程序，比如，SCSI磁带驱动程序用它来确保设备被关闭之前所有的数据都被写入到磁带中。如果*flush*被置为NULL，内核将简单地忽略用户应用程序的请求。

```
int (*release) (struct inode *, struct file *);
```

当file结构被释放时，将调用这个操作。与*open*相仿，也可以将*release*设置为NULL（注5）。

注5：注意，*release*并不是在进程每次调用*close*时都会被调用。只要file结构被共享（如在*fork*或*dup*调用之后），*release*就会等到所有的副本都关闭之后才会得到调用。如果需要在关闭任意一个副本时刷新那些待处理的数据，则应实现*flush*方法。

```
int (*fsync) (struct file *, struct dentry *, int);
```

该方法是*fsync*系统调用的后端实现，用户调用它来刷新待处理的数据。如果驱动程序没有实现这一方法，*fsync*系统调用返回-EINVAL。

```
int (*aio_fsync)(struct kiocb *, int);
```

这是*fsync*方法的异步版本。

```
int (*fasync) (int, struct file *, int);
```

这个操作用来通知设备其FASYNC标志发生了变化。异步通知是比较高级的话题，将在第六章介绍。如果设备不支持异步通知，该字段可以是NULL。

```
int (*lock) (struct file *, int, struct file_lock *);
```

*lock*方法用于实现文件锁定，锁定是常规文件不可缺少的特性，但设备驱动程序几乎从来不会实现这个方法。

```
ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
```

这些方法用来实现分散/聚集型的读写操作。应用程序有时需要进行涉及多个内存区域的单次读或写操作，利用上面这些系统调用可完成这类工作，而不必强加额外的数据拷贝操作。如果这些函数指针被设置为NULL，就会调用*read*和*write*方法（可能是多次）。

```
ssize_t (*sendfile)(struct file *, loff_t *, size_t, read_actor_t, void *);
```

这个方法实现*sendfile*系统调用的读取部分。*sendfile*系统调用以最小的复制操作将数据从一个文件描述符移动到另一个。例如，Web服务器可以利用这个方法将某个文件的内容发送到网络连接。设备驱动程序通常将*sendfile*设置为NULL。

```
ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *,
  int);
```

*sendpage*是*sendfile*系统调用的另外一半，它由内核调用以将数据发送到对应的文件，每次一个数据页。设备驱动程序通常也不需要实现*sendpage*。

```
unsigned long (*get_unmapped_area)(struct file *, unsigned long,
  unsigned long, unsigned long, unsigned long);
```

该方法的目的是在进程的地址空间中找到一个合适的位置，以便将底层设备中的内存段映射到该位置。该任务通常由内存管理代码完成，但该方法的存在可允许驱动程序强制满足特定设备需要的任何对齐需求。大部分驱动程序可设置该方法为NULL。

```
int (*check_flags)(int)
```

该方法允许模块检查传递给*fcntl(F_SETFL...)*调用的标志。

```
int (*dir_notify)(struct file *, unsigned long);
```

当应用程序使用*fcntl*来请求目录改变通知时，该方法将被调用。该方法仅对文件系统有用，驱动程序不必实现*dir_notify*。

*scull*设备驱动程序所实现的只是最重要的设备方法，它的file_operations结构被初始化为如下形式：

```
struct file_operations scull_fops = {
    .owner =    THIS_MODULE,
    .llseek =   scull_llseek,
    .read =     scull_read,
    .write =    scull_write,
    .ioctl =    scull_ioctl,
    .open =     scull_open,
    .release =  scull_release,
};
```

这个声明采用了标准C的标记化结构初始化语法。这种语法是值得采用的，因为它使驱动程序在结构的定义发生变化时更具可移植性，并且使得代码更加紧凑且易读。标记化的初始化方法允许对结构成员进行重新排列。在某些场合下，将频繁被访问的成员放在相同的硬件缓存行上，将大大提高性能。

file 结构

在*<linux/fs.h>*中定义的struct file是设备驱动程序所使用的第二个最重要的数据结构。注意，file结构与用户空间程序中的FILE没有任何关联。FILE在C库中定义且不会出现在内核代码中；而struct file是一个内核结构，它不会出现在用户程序中。

file结构代表一个打开的文件（它并不仅仅限定于设备驱动程序，系统中每个打开的文件在内核空间都有一个对应的file结构）。它由内核在*open*时创建，并传递给在该文件上进行操作的所有函数，直到最后的*close*函数。在文件的所有实例都被关闭之后，内核会释放这个数据结构。

在内核源码中，指向struct file的指针通常被称为file或filp（“文件指针”）。为了不至于和这个结构本身相混淆，我们一致将该指针称为filp。这样，file指的是结构本身，filp则是指向该结构的指针。

struct file中最重要的成员罗列如下。与上节相似，这张清单在首次阅读时可以略过。在下一节中将看到一些真正的C代码，我们会详细讨论其中的某些字段。

```
mode_t f_mode;
```

文件模式。它通过FMODE_READ和FMODE_WRITE位来标识文件是否可读或可写（或

可读写)。读者可能会认为要在自己的*open*或*ioctl*函数中查看这个字段，以便检查是否拥有读/写访问权限，但由于内核在调用驱动程序的*read*和*write*前已经检查了访问权限，所以不必为这两个方法检查权限。在没有获得对应访问权限而打开文件的情况下，对文件的读写操作将被内核拒绝，驱动程序无需为此而作额外的判断。

`loff_t f_pos;`

当前的读/写位置。`loff_t`是一个64位的数(用*gcc*的术语说就是`long long`)。如果驱动程序需要知道文件中的当前位置，可以读取这个值，但不要去修改它。*read*/*write*会使用它们接收到的最后那个指针参数来更新这一位置，而不是直接对`filp->f_pos`进行操作。这一规则的一个例外是*llseek*方法，该方法的目的本身就是为了修改文件位置。

`unsigned int f_flags;`

文件标志，如`O_RDONLY`、`O_NONBLOCK`和`O_SYNC`。为了检查用户请求的是否是非阻塞式的操作(我们将在第六章的“阻塞和非阻塞操作”一节中讨论非阻塞I/O)，驱动程序需要检查`O_NONBLOCK`标志，而其他标志很少用到。注意，检查读/写权限应该查看`f_mode`而不是`f_flags`。所有这些标志都定义在*<linux/fcntl.h>*中。

`struct file_operations *f_op;`

与文件相关的操作。内核在执行*open*操作时对这个指针赋值，以后需要处理这些操作时就读取这个指针。`filp->f_op`中的值决不会为方便引用而保存起来；也就是说，我们可以在任何需要的时候修改文件的关联操作，在返回给调用者之后，新的操作方法就会立即生效。例如，对应于主设备号1(*/dev/null*、*/dev/zero*等等)的*open*代码根据要打开的次设备号替换`filp->f_op`中的操作。这种技巧允许相同主设备号下的设备实现多种操作行为，而不会增加系统调用的负担。这种替换文件操作的能力在面向对象编程技术中称为“方法重载”。

`void *private_data;`

*open*系统调用在调用驱动程序的*open*方法前将这个指针置为`NULL`。驱动程序可以将这个字段用于任何目的或者忽略这个字段。驱动程序可以用这个字段指向已分配的数据，但是一定要在内核销毁`file`结构前在*release*方法中释放内存。`private_data`是跨系统调用时保存状态信息的非常有用的资源，我们的大部分示例都使用了它。

`struct dentry *f_dentry;`

文件对应的目录项(*dentry*)结构。除了用`filp->f_dentry->d_inode`的方式来访问索引节点结构之外，设备驱动程序的开发者们一般无需关心dentry结构。

实际的结构里还有其他一些字段，但它们对于设备驱动程序并没有多大用处。由于驱动程序从不自己填写file结构，而只是对别处创建的file结构进行访问，所以忽略这些字段是安全的。

inode 结构

内核用*inode*结构在内部表示文件，因此它和file结构不同，后者表示打开的文件描述符。对单个文件，可能会有许多个表示打开的文件描述符的file结构，但它们都指向单个inode结构。

inode结构中包含了大量有关文件的信息。作为常规，只有下面两个字段对编写驱动程序代码有用：

`dev_t i_rdev;`

对表示设备文件的inode结构，该字段包含了真正的设备编号。

`struct cdev *i_cdev;`

struct cdev是表示字符设备的内核的内部结构。当inode指向一个字符设备文件时，该字段包含了指向struct cdev结构的指针。

i_rdev的类型在2.5开发系列版本中发生了变化，这破坏了大量驱动程序代码的兼容性。为了鼓励编写可移植性更强的代码，内核开发者增加了两个新的宏，可用来从一个inode中获得主设备号和次设备号：

```
unsigned int iminor(struct inode *inode);
unsigned int imajor(struct inode *inode);
```

为了防止因为类似的改变而出现问题，我们应该使用上述宏，而不是直接操作i_rdev。

字符设备的注册

我们前面提到，内核内部使用struct cdev结构来表示字符设备。在内核调用设备的操作之前，必须分配并注册一个或者多个上述结构（注6）。为此，我们的代码应包含*<linux/cdev.h>*，其中定义了这个结构以及与其相关的一些辅助函数。

分配和初始化上述结构的方式有两种。如果读者打算在运行时获取一个独立的cdev结构，则应该如下编写代码：

注6：有一个老的机制可避免使用cdev结构（我们将在“老方法”一节中讨论）。但是，新代码应使用新的技术。

```
struct cdev *my_cdev = cdev_alloc();
my_cdev->ops = &my_fops;
```

这时，你可以将cdev结构嵌入到自己的设备特定结构中，*scull*就是这样做的。这种情况下，我们需要用下面的代码来初始化已分配到的结构：

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

另外，还有一个struct cdev的字段需要初始化。和file_operations结构类似，struct cdev也有一个所有者字段，应被设置为THIS_MODULE。

在cdev结构设置好之后，最后的步骤是通过下面的调用告诉内核该结构的信息：

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

这里，dev是cdev结构，num是该设备对应的第一个设备编号，count是应该和该设备关联的设备编号的数量。count经常取1，但是在某些情形下，会有多个设备编号对应于一个特定的设备。例如，考虑SCSI磁带驱动程序，它通过每个物理设备的多个次设备号来允许用户空间选择不同的操作模式（比如密度）。

在使用*cdev_add*时，需要牢记重要的一点。首先，这个调用可能会失败。如果它返回一个负的错误码，则设备不会被添加到系统中。但这个调用几乎总会成功返回，此时，我们又面临另一个问题：只要*cdev_add*返回了，我们的设备就“活”了，它的操作就会被内核调用。因此，在驱动程序还没有完全准备好处理设备上的操作时，就不能调用*cdev_add*。

要从系统中移除一个字符设备，做如下调用：

```
void cdev_del(struct cdev *dev);
```

要清楚的是，在将cdev结构传递到*cdev_del*函数之后，就不应再访问cdev结构了。

Scull中的设备注册

在*scull*内部，它通过struct scull_dev的结构来表示每个设备，该结构定义如下：

```
struct scull_dev {
    struct scull_qset *data;  /* 指向第一个量子集的指针 */
    int quantum;              /* 当前量子的大小 */
    int qset;                 /* 当前数组的大小 */
    unsigned long size;       /* 保存在其中的数据总量 */
    unsigned int access_key;  /* 由sculluid和scullpriv使用 */
    struct semaphore sem;     /* 互斥信号量 */
    struct cdev cdev;         /* 字符设备结构       */
};
```

我们会在遇到该结构字段的时候讨论它们，而现在，我们的注意力集中在cdev上，即内核和设备间的接口struct cdev。该结构必须如上所述地被初始化并添加到系统中，*scull*中完成这一工作的代码如下：

```
static void scull_setup_cdev(struct scull_dev *dev, int index)
{
    int err, devno = MKDEV(scull_major, scull_minor + index);

    cdev_init(&dev->cdev, &scull_fops);
    dev->cdev.owner = THIS_MODULE;
    dev->cdev.ops = &scull_fops;
    err = cdev_add (&dev->cdev, devno, 1);
    /* Fail gracefully if need be */
    if (err)
    printk(KERN_NOTICE "Error %d adding scull%d", err, index);
}
```

因为cdev结构被嵌入到了struct scull_dev中，因此必须调用*cdev_init*来执行该结构的初始化。

早期的办法

如果读者阅读2.6内核中的其他驱动程序代码，也许会注意到相当数量的字符设备驱动程序不使用我们前面描述过的cdev接口。其实读者看到的是尚未升级到2.6接口的老代码。因为这些代码也可以工作，因此在较长时间内升级可能不会发生。为了完整起见，我们将描述老的字符设备注册接口，但新的代码不应该使用这些老的接口，因为这种机制会在将来的内核中消失。

注册一个字符设备驱动程序的经典方式是：

```
int register_chrdev(unsigned int major, const char *name,
                    struct file_operations *fops);
```

这里，major是设备的主设备号，name是驱动程序的名称（出现在*/proc/devices*中），而fops是默认的file_operations结构。对*register_chrdev*的调用将为给定的主设备号注册0～255作为次设备号，并为每个设备建立一个对应的默认cdev结构。使用这一接口的驱动程序必须能够处理所有256个次设备号上的*open*调用（不论它们是否真正对应于实际的设备），而且也不能使用大于255的主设备号和次设备号。

如果使用*register_chrdev*函数，将自己的设备从系统中移除的正确函数是：

```
int unregister_chrdev(unsigned int major, const char *name);
```

major和name必须与传递给*register_chrdev*函数的值保持一致，否则该调用会失败。

open 和 release

现在我们已经简单地浏览了这些字段，下面将在实际的 *scull* 函数中使用这些字段。

open 方法

open 方法提供给驱动程序以初始化的能力，从而为以后的操作完成初始化做准备。在大部分驱动程序中，*open* 应完成如下工作：

- 检查设备特定的错误（诸如设备未就绪或类似的硬件问题）。
- 如果设备是首次打开，则对其进行初始化。
- 如有必要，更新 f_op 指针。
- 分配并填写置于 filp->private_data 里的数据结构。

然而，首先要做的就是确定要打开的具体设备。注意，*open* 方法的原型如下：

```
int (*open)(struct inode *inode, struct file *filp);
```

其中的 *inode* 参数在其 i_cdev 字段中包含了我们所需要的信息，即我们先前设置的 cdev 结构。唯一的问题是，我们通常不需要 cdev 结构本身，而是希望得到包含 cdev 结构的 scull_dev 结构。C 语言可帮助程序员通过一些技巧完成这类转换，但不应滥用这类技巧，它会使得代码对其他人来讲难于阅读和理解。幸运的是，在这种情况下内核黑客已经帮助我们实现了此类技巧，它通过定义在 *<linux/kernel.h>* 中的 *container_of* 宏实现：

```
container_of(pointer, container_type, container_field);
```

这个宏需要一个 container_field 字段的指针，该字段包含在 container_type 类型的结构中，然后返回包含该字段的结构指针。在 *scull_open* 中，这个宏用来找到适当的设备结构：

```
struct scull_dev *dev; /* device information */

dev = container_of(inode->i_cdev, struct scull_dev, cdev);
filp->private_data = dev; /* for other methods */
```

一旦代码找到 scull_dev 结构之后，*scull* 将一个指针保存到了 file 结构的 private_data 字段中，这样可以方便今后对该指针的访问。

另外一个确定要打开的设备的方法是：检查保存在 inode 结构中的次设备号。如果读者利用 *register_chrdev* 注册自己的设备，则必须使用该技术。请一定使用 *iminor* 宏从 inode 结构中获得次设备号，并确保它对应于驱动程序真正准备打开的设备。

```
The (slightly simplified) code for scull_open is:
```

经过些微简化的*scull_open*代码如下:

```
int scull_open(struct inode *inode, struct file *filp)
{
    struct scull_dev *dev; /* device information */

    dev = container_of(inode->i_cdev, struct scull_dev, cdev);
    filp->private_data = dev; /* for other methods */

    /* now trim to 0 the length of the device if open was write-only */
    if ( (filp->f_flags & O_ACCMODE) =  = O_WRONLY) {
        scull_trim(dev); /* ignore errors */
    }
    return 0;          /* success */
}
```

这段代码看起来相当短小，因为在调用*open*时它并没有做针对某个特定设备的任何处理。由于*scull*设备被设计为全局且持久的，这段代码无需做什么工作。特别是，由于我们并不维护*scull*的打开计数，而只维护模块的使用计数，因此也就没有类似于“首次打开时初始化设备”的这类动作。

对设备唯一的实际操作是，当设备以写方式打开时，它的长度将被截为0。出现这种特性的原因在于，在设计上，当用更短的文件覆盖一个*scull*设备时，设备数据区应相应缩小。这与用写入方式打开普通文件时将长度截短为0的方式很相似。如果设备以读取方式打开，则什么也不做。

稍后在浏览其他*scull*设备类型的个性代码时，我们将会看到真正的初始化工作是如何完成的。

release 方法

*release*方法的作用正好与*open*相反。有时读者会发现这个方法的实现被称为*device_*close，而不是*device_*release。无论是哪种形式，这个设备方法都应该完成下面的任务:

- 释放由*open*分配的、保存在filp->private_data中的所有内容。
- 在最后一次关闭操作时关闭设备。

*scull*的基本模型没有需要关闭的硬件，因此所需的代码量最少（注7):

注7: 因为*scull_open*为每种设备都替换了不同的filp->f_op，所以不同的设备由不同的函数关闭。我们随后会讨论这些内容。

```
int scull_release(struct inode *inode, struct file *filp)
{
    return 0;
}
```

当关闭一个设备文件的次数比打开它的次数多时，系统中会发生什么情况呢？毕竟，*dup*和*fork*系统调用都会在不调用*open*的情况下创建已打开文件的副本，但每一个副本都会在程序终止时被关闭。例如，大多数程序从来不打开它们的*stdin*文件（或设备），但它们都会在终止时被关闭。那么，驱动程序如何才能知道一个打开的设备文件要被真正关闭呢？

答案很简单：并不是每个*close*系统调用都会引起对*release*方法的调用。只有那些真正释放设备数据结构的*close*调用才会调用这个方法。内核对每个`file`结构维护其被使用多少次的计数器。无论是*fork*还是*dup*，都不会创建新的数据结构（仅由*open*创建），它们只是增加已有结构中的计数。只有在`file`结构的计数归0时，*close*系统调用才会执行*release*方法，这只在删除这个结构时才会发生。*release*方法与*close*系统调用间的关系保证了对于每次*open*驱动程序只会看到对应的一次*release*调用。

注意：*flush*方法在应用程序每次调用*close*时都会被调用。不过，很少有驱动程序会去实现*flush*，因为在close时并没有什么事情需要去做，除非*release*被调用。

正如读者猜想的那样，甚至在应用程序还未显式地关闭它所打开的文件就终止时，以上的讨论同样也是适用的：内核在进程退出的时候，通过在内部使用*close*系统调用自动关闭所有相关的文件。

scull的内存使用

在介绍*read*和*write*操作以前，我们最好先看看*scull*如何并且为何进行内存分配。为了彻底理解代码，我们需要知道“如何分配”，而“为何分配”则表明了驱动程序编写者所需做出的选择，尽管*scull*作为设备来说肯定还不具备代表性。

本节只讲解*scull*中的内存分配策略，而不会涉及编写实际驱动程序时所需要的硬件管理技巧。这些技巧将在第九章和第十章中介绍。因此，如果读者对面向内存操作的*scull*驱动程序的内部工作原理不感兴趣的话，可以跳过这一节。

*scull*使用的内存区域这里也称为设备，其长度是可变的。写得越多，它就变得越长；用更短的文件以覆盖方式写设备时则会变短。

*scull*驱动程序引入了Linux内核中用于内存管理的两个核心函数。这两个函数定义在*<linux/slab.h>*中，它们是：

```
void *kmalloc(size_t size, int flags);
void kfree(void *ptr);
```

对*kmalloc*的调用将试图分配size个字节大小的内存；其返回值指向该内存的指针，分配失败时返回NULL。flags参数用来描述内存的分配方法，我们将在第八章详细描述这些标志。到目前为止，我们始终使用GFP_KERNEL。由这个函数分配的内存应该通过*kfree*释放。我们不应该将非*kmalloc*返回的指针传递给*kfree*。但是，将NULL指针传递给*kfree*是合法的。

对分配大的内存区域来说，*kmalloc*并不是最有效的方法（参见第八章），因此*scull*的实现方法并不是很巧妙。但更为巧妙的实现其代码读起来会较困难，而本节的目的只是讲解*read*和*write*，并非内存管理。这也就是为什么虽然分配整个页面会更有效，但代码只使用了*kmalloc*和*kfree*，而没有采取分配整个页面的操作的原因。

另一方面，从理论和实际的角度考虑，我们不想限制“设备”的尺寸。从理论角度来看，对所管理的数据项任意增加限制总是很糟糕的想法。从实际角度来看，为了在内存短缺的情况下进行测试，可利用*scull*暂时将系统的内存吃光。进行这样的测试有助于了解系统内部。可以使用命令*cp /dev/zero /dev/scull0*用光所有的系统RAM，也可以用*dd*工具选择复制多少数据到*scull*设备中。

在*scull*中，每个设备都是一个指针链表，其中每个指针都指向一个scull_qset结构。默认情况下，每一个这样的结构通过一个中间指针数组最多可引用4 000 000个字节。我们发布的源代码使用了一个有1000个指针的数组，每个指针指向一个4000字节的区域。我们把每一个内存区称为一个量子，而这个指针数组（或它的长度）称为量子集。*scull*设备和它的内存区如图3-1所示。

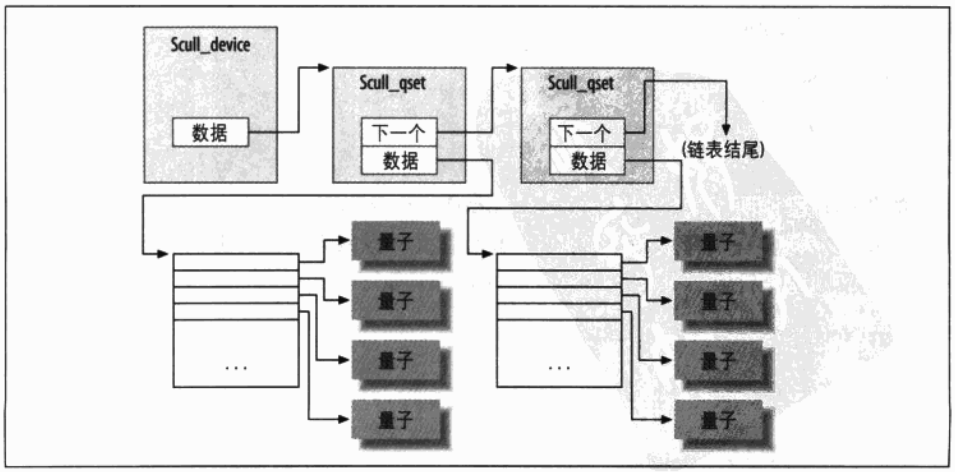


图3-1：scull设备的布局

这样，选择的参数使得向*scull*写入一个字节就会消耗8000或12000个字节的内存：每个量子占用4000个字节，而一个量子集占用4000或8000个字节（取决于目标平台上的指针本身占用32位还是64位）。然而，如果向*scull*写入大量的数据，链表的开支并不会太大。每4MB数据只对应一个链表元素，而设备的最大尺寸受计算机内存大小的限制。

为量子和量子集选择合适的数值是一个策略问题而非机制问题，而且最优数值依赖于如何使用设备。因此*scull*设备的驱动程序不应对量子和量子集的尺寸强制使用某个特定的数值。在*scull*设备中，用户可以采用几种方式来修改这些值：在编译时，可以修改*scull.h*中的宏SCULL_QUANTUM和SCULL_QSET；而在模块加载时，可以设置scull_quantum和scull_qset的整数值；或者在运行时，使用*ioctl*修改当前值以及默认值。

使用宏和整数值同时允许在编译期间和加载阶段进行配置，这种方法和前面选择主设备号的方法类似。对于驱动程序中任何不确定的或与策略相关的数值，我们都可以使用这种技巧。

余下的唯一问题是如何选择默认数值。在这个例子里，量子和量子集未充分填满会导致内存浪费，而量子和量子集过小则会在进行内存分配、释放和指针链接等操作时增加系统开销，默认数值的选择问题就在于寻找这两者之间的最佳平衡点。此外，还必须考虑*kmalloc*的内部设计，然而目前我们还无法涉及这一点。*kmalloc*的内部结构将在第八章中探讨。默认数值的选择基于这样的假设：在测试*scull*时，可能会有大块的数据写入其中，但大多数情况下，对该设备的正常使用可能只传递几KB的数据量。

我们已经看到内部表示*scull*设备的scull_dev结构。该结构的quantum和qset字段分别保存设备的量子和量子集大小。但是，实际的数据由另外的结构处理，该结构称为struct scull_qset：

```
struct scull_qset {
    void **data;
    struct scull_qset *next;
};
```

下面的代码片段说明了如何利用struct scull_dev和struct scull_qset保存数据。*scull_trim*函数负责释放整个数据区，并且在文件以写入方式打开时由*scull_open*调用。它简单地遍历链表，并释放所有找到的量子和量子集。

```
int scull_trim(struct scull_dev *dev)
{
    struct scull_qset *next, *dptr;
    int qset = dev->qset;   /* "dev" 非空 */
    int i;
    for (dptr = dev->data; dptr; dptr = next) { /* 所有链表项 */
```

```
            if (dptr->data) {
                for (i = 0; i < qset; i++)
                    kfree(dptr->data[i]);
                kfree(dptr->data);
                dptr->data = NULL;
            }
            next = dptr->next;
            kfree(dptr);
        }
        dev->size = 0;
        dev->quantum = scull_quantum;
        dev->qset = scull_qset;
        dev->data = NULL;
        return 0;
    }
```

模块的清除函数也调用 *scull_trim* 函数，以便将由 *scull* 所使用的内存返回给系统。

read 和 write

read 和 *write* 方法完成的任务是相似的，亦即，拷贝数据到应用程序空间，或反过来从应用程序空间拷贝数据。因此，它们的原型很相似，不妨同时介绍它们：

```
ssize_t read(struct file *filp, char _ _user *buff,
    size_t count, loff_t *offp);
ssize_t write(struct file *filp, const char _ _user *buff,
    size_t count, loff_t *offp);
```

对于这两个方法，参数 `filp` 是文件指针，参数 `count` 是请求传输的数据长度。参数 `buff` 是指向用户空间的缓冲区，这个缓冲区或者保存要写入的数据，或者是一个存放新读入数据的空缓冲区。最后的 `offp` 是一个指向“long offset type（长偏移量类型）”对象的指针，这个对象指明用户在文件中进行存取操作的位置。返回值是“signed size type（有符号的尺寸类型）”，后面会谈到它的用法。

需要再次指出的是，*read* 和 *write* 方法的 `buff` 参数是用户空间的指针。因此，内核代码不能直接引用其中的内容。出现这种限制的原因有如下几个：

- 随着驱动程序所运行的架构的不同或者内核配置的不同，在内核模式中运行时，用户空间的指针可能是无效的。该地址可能根本无法被映射到内核空间，或者可能指向某些随机数据。
- 即使该指针在内核空间中代表相同的东西，但用户空间的内存是分页的，而在系统调用被调用时，涉及到的内存可能根本就不在RAM中。对用户空间内存的直接引用将导致页错误，而这对内核代码来说是不允许发生的事情。其结果可能是一个“oops”，它将导致调用该系统调用的进程死亡。

- 我们讨论的指针可能由用户程序提供，而该程序可能存在缺陷或者是个恶意程序。如果我们的驱动程序盲目引用用户提供的指针，将导致系统出现打开的后门，从而允许用户空间程序随意访问或覆盖系统中的内存。如果读者不打算因为自己的驱动程序而危及用户系统的安全性，则永远不应直接引用用户空间指针。

很显然，驱动程序必须访问用户空间的缓冲区以便完成自己的工作。为了确保安全，这种访问应始终通过内核提供的专用函数完成。我们将在这里介绍其中几个这类函数（在*<asm/uaccess.h>*中定义），其他的函数将在第六章的“使用ioctl的参数”一节中讲述，这些函数使用了一些特殊的、架构相关的方法来确保在内核和用户空间之间安全、正确地交换数据。

*scull*的*read*和*write*代码要做的工作就是在用户地址空间和内核地址空间之间进行整段数据的拷贝。这种能力是由下面的内核函数提供的，它们用于拷贝任意的一段字节序列，这也是大多数*read*和*write*方法实现的核心部分：

```
unsigned long copy_to_user(void _ _user *to,
                           const void *from,
                           unsigned long count);
unsigned long copy_from_user(void *to,
                             const void _ _user *from,
                             unsigned long count);
```

虽然这些函数的行为很像通常的*memcpy*函数，但当内核空间内运行的代码访问用户空间时要多加小心。被寻址的用户空间的页面可能当前并不在内存中，于是虚拟内存子系统会将该进程转入睡眠状态，直到该页面被传送至期望的位置。例如，当页面必须从交换空间取回时，这样的情况就会发生。对于驱动程序编写人员来说，这带来的结果就是访问用户空间的任何函数都必须是可重入的，并且必须能和其他驱动程序函数并发执行，更特别的是，必须处于能够合法休眠的状态。我们将在第五章详细讨论相关话题。

这两个函数的作用并不限于在内核空间和用户空间之间拷贝数据，它们还检查用户空间的指针是否有效。如果指针无效，就不会进行拷贝；另一方面，如果在拷贝过程中遇到无效地址，则仅仅会复制部分数据。在这两种情况下，返回值是还需要拷贝的内存数量值。*scull*代码如果发现这样的错误返回（即返回值不为0时），会给用户返回-EFAULT。

关于用户空间访问和无效用户空间指针的内容是相对高级的话题，第六章会对它们进行进一步讨论。如果并不需要检查用户空间指针，那么建议读者转而调用*__copy_to_user*和*__copy_from_user*。在预先知道参数已经检查过时，这两个函数还是很有用的。但要小心的是，如果我们并没有真正检查传递给这些函数的用户空间指针，则可能会导致内核崩溃和/或建立安全漏洞。

至于实际的设备方法，*read* 方法的任务是从设备拷贝数据到用户空间（使用 *copy_to_user*），而 *write* 方法则是从用户空间拷贝数据到设备上（使用 *copy_from_user*）。每次 *read* 或 *write* 系统调用都会请求一定数目的字节传输，不过驱动程序也并不限制小数据量的传输 —— 读 / 写之间的确切规则还是有些细微差异的，本章后面的内容中会提到。

无论这些方法传输了多少数据，一般而言都应更新 `*offp` 所表示的文件位置，以便反映在新系统调用成功完成之后当前的文件位置。适当情况下，内核会将文件位置的改变传播回 `file` 结构。但是，*pread* 和 *pwrite* 系统调用具有不同的语义；这两个系统调用从一个给定的文件偏移量开始操作，并且不会修改文件位置。它们会传入一个指针，该指针指向用户提供的位置，而且会丢弃驱动程序所作的任何修改。

图 3-2 表明了一个典型的 *read* 实现是如何使用其参数的。

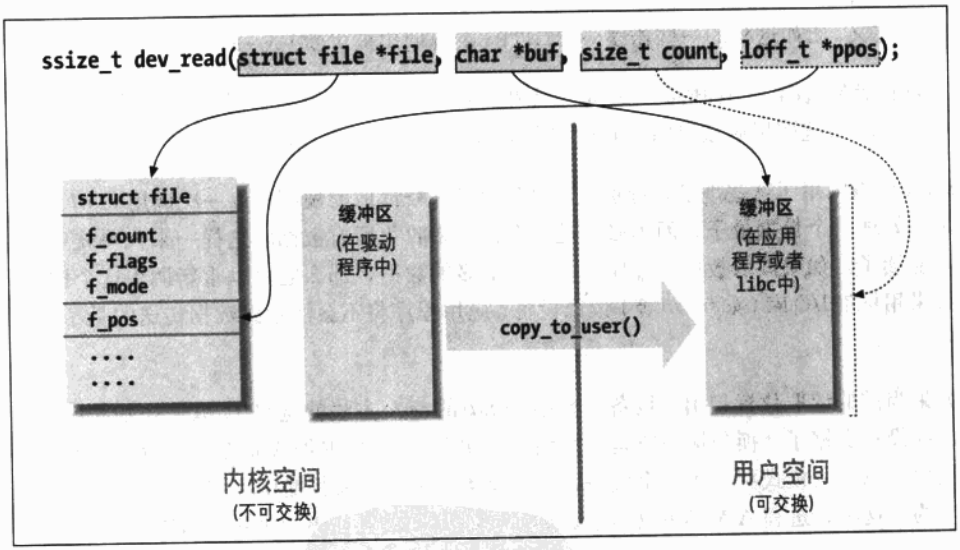


图 3-2：read 的参数

出错时，*read* 和 *write* 方法都返回一个负值。大于等于 0 的返回值告诉调用程序成功传输了多少字节。如果在正确传输部分数据之后发生了错误，则返回值必须是成功传输的字节数，但这个错误只能在下一次函数调用时才会得到报告。当然，这种实现惯例要求驱动程序必须记住错误的发生，这样才能在将来把错误状态返回给应用程序。

尽管内核函数通过返回负值来表示错误，而且该返回值表明了错误的类型（见第二章），但运行在用户空间的程序看到的始终是作为返回值的 `-1`。为了找到出错原因，用户空间的程序必须访问 `errno` 变量。用户空间的这种行为源于 POSIX 标准，但该标准并未对内核内部的操作做任何要求。

read 方法

调用程序对 *read* 的返回值解释如下:

- 如果返回值等于传递给 *read* 系统调用的 count 参数，则说明所请求的字节数传输成功完成了。这是最理想的情况。
- 如果返回值是正的，但是比 count 小，则说明只有部分数据成功传送。这种情况因设备的不同可能有许多原因。大部分情况下，程序会重新读数据。例如，如果用 *fread* 函数读数据，这个库函数就会不断调用系统调用，直至所请求的数据传输完毕为止。
- 如果返回值为 0，则表示已经到达了文件尾。
- 负值意味着发生了错误，该值指明了发生了什么错误，错误码在<*linux/errno.h*>中定义。比如这样的一些错误: -EINTR(系统调用被中断)或-EFAULT(无效地址)。

上面的清单遗漏了一种情况，就是“现在还没有数据，但以后可能会有”。在这种情况下，*read* 系统调用应该阻塞。我们将在第六章中讲述阻塞读取。

scull 代码利用了这些规则，特别地，它利用了部分读取的规则。每一次调用 *scull_read* 时只处理一个数据量子，而不必通过一个循环来收集所有数据；这样一来代码就更短、更易读了。如果读取数据的程序确实需要更多的数据，那么它可以重新调用这个调用。如果用标准I/O库（如*fread*等）读取设备，应用程序将不会注意到数据传送的量子化过程。

如果当前的读取位置超出了设备的大小，*scull* 的 *read* 方法就返回 0，以便告知程序这里已经没有数据了（换句话说就是已经到文件尾了）。如果进程 A 正在读取设备，而此时进程 B 以写入模式打开了这个设备，于是设备会被截断为长度 0，这种情况是有可能发生的。这时，进程 A 突然发现自己超过了文件尾，并且在下次调用 *read* 时返回 0。

下面是 *read* 的代码（忽略了对 *down_interruptible* 调用，对此我们将在下一章介绍）:

```
ssize_t scull_read(struct file *filp, char __user *buf, size_t count,
                   loff_t *f_pos)
{
    struct scull_dev *dev = filp->private_data;
    struct scull_qset *dptr;    /* 第一个链表项 */
    int quantum = dev->quantum, qset = dev->qset;
    int itemsize = quantum * qset; /* 该链表项中有多少字节 */
    int item, s_pos, q_pos, rest;
    ssize_t retval = 0;

    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
```

```
    if (*f_pos >= dev->size)
        goto out;
    if (*f_pos + count > dev->size)
        count = dev->size - *f_pos;

    /* 在量子集中寻找链表项、qset 索引以及偏移量 */
    item = (long)*f_pos / itemsize;
    rest = (long)*f_pos % itemsize;
    s_pos = rest / quantum; q_pos = rest % quantum;

    /* 沿该链表前行，直到正确的位置（在其他地方定义）*/
    dptr = scull_follow(dev, item);

    if (dptr =  = NULL || !dptr->data || ! dptr->data[s_pos])
        goto out; /* don't fill holes */

    /* 读取该量子的数据直到结尾 */
    if (count > quantum - q_pos)
        count = quantum - q_pos;

    if (copy_to_user(buf, dptr->data[s_pos] + q_pos, count)) {
        retval = -EFAULT;
        goto out;
    }
    *f_pos += count;
    retval = count;

  out:
    up(&dev->sem);
    return retval;
}
```

write 方法

与 *read* 类似，根据如下返回值规则，*write* 也能传输少于请求的数据量：

- 如果返回值等于 count，则完成了所请求数目的字节传送。
- 如果返回值是正的，但小于 count，则只传输了部分数据。程序很可能再次试图写入余下的数据。
- 如果值为 0，意味着什么也没写入。这个结果不是错误，而且也没有理由返回一个错误码。再次重申，标准库会重复调用 *write*。在第六章介绍阻塞式 *write* 时，我们将详细说明这种情形。
- 负值意味发生了错误，与 *read* 相同，有效的错误码定义在 *<linux/errno.h>* 中。

不幸的是，有些错误程序只进行了部分传输就报错并异常退出。这种情况的发生是由于程序员习惯于认定 *write* 调用要么失败要么就完全成功。在大多数时候的确是这样的，设

备驱动也应对此进行支持。这种局限性在*scull*的实现中可以弥补，但我们不想把代码搞得太复杂，能说明问题就行了。

与*read*方法一样，*scull*的*write*代码每次只处理一个量子：

```
ssize_t scull_write(struct file *filp, const char _ _user *buf, size_t count,
                loff_t *f_pos)
{
    struct scull_dev *dev = filp->private_data;
    struct scull_qset *dptr;
    int quantum = dev->quantum, qset = dev->qset;
    int itemsize = quantum * qset;
    int item, s_pos, q_pos, rest;
    ssize_t retval = -ENOMEM; /*  "goto out"语句使用的值 */

    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;

    /* 在量子集中寻找链表项、qset 索引以及偏移量 */
    item = (long)*f_pos / itemsize;
    rest = (long)*f_pos % itemsize;
    s_pos = rest / quantum; q_pos = rest % quantum;

    /* 沿该链表前行，直到正确的位置（在其他地方定义）*/
    dptr = scull_follow(dev, item);
    if (dptr =  = NULL)
        goto out;
    if (!dptr->data) {
        dptr->data = kmalloc(qset * sizeof(char *), GFP_KERNEL);
        if (!dptr->data)
            goto out;
        memset(dptr->data, 0, qset * sizeof(char *));
    }
    if (!dptr->data[s_pos]) {
        dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
        if (!dptr->data[s_pos])
            goto out;
    }
    /* 将数据写入该量子，直到结尾 */
    if (count > quantum - q_pos)
        count = quantum - q_pos;

    if (copy_from_user(dptr->data[s_pos]+q_pos, buf, count)) {
        retval = -EFAULT;
        goto out;
    }
    *f_pos += count;
    retval = count;

        /* 更新文件大小 */
    if (dev->size < *f_pos)
        dev->size = *f_pos;
```

```
  out:
    up(&dev->sem);
    return retval;
}
```

readv 和 writev

Unix 系统很早就已支持两个可选的系统调用：*readv* 和 *writev*。这些“向量”型的函数具有一个结构数组，每个结构包含一个指向缓冲区的指针和一个长度值。*readv* 调用可用于将指定数量的数据依次读入每个缓冲区。*writev*则是把各个缓冲区的内容收集起来，并将它们在一次写入操作中进行输出。

如果驱动程序没有提供用于处理向量操作的方法，*readv*和*writev*会通过对*read*和*write*方法的多次调用来实现。但在很多情况下，直接在驱动程序中实现 *readv* 和 *writev* 可以获得更高的效率。

向量操作的函数原型如下：

```
ssize_t (*readv) (struct file *filp, const struct iovec *iov,
                  unsigned long count, loff_t *ppos);
ssize_t (*writev) (struct file *filp, const struct iovec *iov,
                  unsigned long count, loff_t *ppos);
```

其中，filp和ppos参数与*read*和*write*方法中的用法相同。iovec结构定义在*<linux/uio.h>*中，其形式如下：

```
struct iovec
{
    void _ _user *iov_base;
    _ _kernel_size_t iov_len;
};
```

每个iovec结构都描述了一个用于传输的数据块——这个数据块的起始位置在iov_base（在用户空间中），长度为iov_len个字节。函数中的count参数指明要操作多少个iovec结构。这些结构由应用程序创建，而内核在调用驱动程序之前会把它们拷贝到内核空间。

向量化操作最简单的实现，可能就是只传递每个iovec结构的地址和长度给驱动程序的*read*或*write*函数。不过，正确而有效率的操作经常需要驱动程序做一些更为巧妙的事情。例如，磁带驱动程序的*writev*就应将所有iovec结构的内容作为磁带上的单个记录写入。

但是，很多驱动程序并不期望通过自己实现这些方法来获益。所以，*scull*忽略了它们。内核将会通过*read*和*write*来模拟它们，而最终结果仍然如此。

试试新设备

一旦准备好了刚才讲述的四个方法，就可以编译和测试驱动程序了，它保留写入的数据，直至用新数据覆盖它们。这个设备有点像长度只受物理RAM容量限制的数据缓冲区。可以试试用 *cp*、*dd* 或者输入/输出重定向等命令来测试这个驱动程序。

依据写入 *scull* 的数据量，用 *free* 命令可以看到空闲内存的缩减和扩增。

为了进一步证实每次是否读写一个量子，可以在驱动程序的适当位置加入 *printk*，从而可了解到程序读/写大数据块时会发生什么事情。此外，还可以用工具 *strace* 来监视应用程序调用的系统调用以及它们的返回值。跟踪 *cp* 或 *ls -l > /dev/scull0* 会显示出量子化的读写过程。第四章将会详细介绍监视（或跟踪）技术。

快速参考

本章介绍了下列符号和头文件。file_operations结构和file结构的字段清单并没有在这里给出。

```
#include <linux/types.h>
```

```
dev_t
```

dev_t 是内核中用来表示设备编号的数据类型。

```
int MAJOR(dev_t dev);
int MINOR(dev_t dev);
```

这两个宏从设备编号中抽取出主/次设备号。

```
dev_t MKDEV(unsigned int major, unsigned int minor);
```

这个宏由主/次设备号构造一个 dev_t 数据项。

```
#include <linux/fs.h>
```

“文件系统”头文件，它是编写设备驱动程序必需的头文件，其中声明了许多重要的函数和数据结构。

```
int register_chrdev_region(dev_t first, unsigned int count, char *name)
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned
  int count, char *name)
void unregister_chrdev_region(dev_t first, unsigned int count);
```

提供给驱动程序用来分配和释放设备编号范围的函数。在期望的主设备号预先知道的情况下，应调用 *register_chrdev_region*；而对动态分配，使用 *alloc_chrdev_region*。

```
int register_chrdev(unsigned int major, const char *name, struct
  file_operations *fops);
```

老的（2.6之前的）字符设备注册例程。2.6内核也提供了仿效该例程的函数，但是新代码不应该再使用该函数。如果主设备号不是0，则不加修改地使用；否则，系统将为该设备动态地分配编号。

```
int unregister_chrdev(unsigned int major, const char *name);
```

用于注销由 *register_chrdev* 函数注册的驱动程序。major 和 name 字符串必须包含与注册该驱动程序时使用的相同的值。

```
struct file_operations;
struct file;
struct inode;
```

大多数设备驱动程序都会用到的三个重要数据结构。file_operations结构保存了字符驱动程序的方法；struct file表示一个打开的文件，而struct inode表示一个磁盘上的文件。

```
#include <linux/cdev.h>
struct cdev *cdev_alloc(void);
void cdev_init(struct cdev *dev, struct file_operations *fops);
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
void cdev_del(struct cdev *dev);
```

用来管理 cdev 结构的函数，内核中使用该结构表示字符设备。

```
#include <linux/kernel.h>
container_of(pointer, type, field);
```

一个方便使用的宏，它可用于从包含在某个结构中的指针获得结构本身的指针。

```
#include <asm/uaccess.h>
```

该头文件声明了在内核代码和用户空间之间移动数据的函数。

```
unsigned long copy_from_user (void *to, const void *from, unsigned
  long count);
unsigned long copy_to_user (void *to, const void *from, unsigned long count);
```

在用户空间和内核空间之间拷贝数据。

第四章

调试技术



内核编程有其自身独特的调试难题。由于内核是一个不与特定进程相关的功能集合，所以内核代码无法轻易地放在调试器中执行，而且也很难跟踪。同样，要想重现内核代码中的错误也是相当困难的，因为这种错误可能导致整个系统崩溃，这样也就破坏了可以用来跟踪它们的现场。

本章将介绍在这种令人痛苦的环境下监视内核代码并跟踪错误的技术。

内核中的调试支持

在第二章，我们建议读者构造并安装自己的内核，而不是运行发行版自带的原有内核。运行自己内核的一个最重要的原因之一是因为内核开发者已经在内核中建立了多项用于调试的功能。但这些功能会造成额外的输出，并导致性能下降，因此发行版厂商通常会禁止发行版内核中的这些功能。但是作为一名内核开发者，调试需求具有更高优先级，从而应该乐意接受因为额外的调试支持而导致的（最小）系统负载。

这里，我们列出了用于内核开发的几个配置选项。除特别指出外，所有这些选项均出现在内核配置工具的“kernel hacking”菜单中。注意，并非所有体系架构都支持其中的某些选项。

CONFIG_DEBUG_KERNEL
: 该选项仅仅使得其他的调试选项可用。我们应该打开该选项，但它本身不会打开所有的调试功能。

CONFIG_DEBUG_SLAB
: 这是一个非常重要的选项，它打开内核内存分配函数中的多个类型的检查；打开该检查后，就可以检测许多内存溢出及忘记初始化的错误。在将已分配内存返回给调

用者之前，内核将把其中的每个字节设置为0xa5，而在释放后将其设置为0x6b。如果读者在自己驱动程序的输出中，或者在oops信息中看到上述“毒剂”字符，则可以轻松判断问题所在。在打开该调试选项后，内核还会在每个已分配内存对象的前面和后面放置一些特殊的防护值；这样，当这些防护值发生变化时，内核就可以知道有些代码超出了内存的正常访问范围，并“大声抱怨”。同时，该选项还会检查更多隐蔽的错误。

CONFIG_DEBUG_PAGEALLOC

在释放时，全部内存页从内核地址空间中移出。该选项将大大降低运行速度，但可以快速定位特定的内存损坏错误的所在位置。

CONFIG_DEBUG_SPINLOCK

打开该选项，内核将捕获对未初始化自旋锁的操作，也会捕获诸如两次解开同一锁的操作等其他错误。

CONFIG_DEBUG_SPINLOCK_SLEEP

该选项将检查拥有自旋锁时的休眠企图。实际上，如果调用可能引起休眠的函数，这个选项也会生效，即使该函数可能不会导致真正的休眠。

CONFIG_INIT_DEBUG

标记为__init（或者__initdata）的符号将会在系统初始化或者模块装载之后被丢弃。该选项可用来检查初始化完成之后对用于初始化的内存空间的访问企图。

CONFIG_DEBUG_INFO

该选项将使内核的构造包含完整的调试信息。如果读者打算利用*gdb*调试内核，将需要这些信息。如果计划使用*gdb*，还应该打开CONFIG_FRAME_POINTER选项。

CONFIG_MAGIC_SYSRQ

打开“SysRq魔法（magic SysRq）”按键。我们将在本章后面的“系统挂起”一节中讲述该按键。

CONFIG_DEBUG_STACKOVERFLOW

CONFIG_DEBUG_STACK_USAGE

这些选项可帮助跟踪内核栈的溢出问题。栈溢出的确切信号是不包含任何合理的反向跟踪信息的oops清单。第一个选项将在内核中增加明确的溢出检查；而第二个选项将让内核监视栈的使用，并通过SysRq按键输出一些统计信息。

CONFIG_KALLSYMS

该选项出现在“General setup/Standard features（一般设置/标准功能）”菜单中，将在内核中包含符号信息；该选项默认是打开的。该符号信息用于调试上下文；没有此符号，oops清单只能给出十六进制的内核反向跟踪信息，这通常没有多少用处。

CONFIG_IKCONFIG

CONFIG_IKCONFIG_PROC

这些选项出现在“General setup（一般设置）”菜单中，会让完整的内核配置状态包含到内核中，并可通过*/proc*访问。大多数内核开发者清楚地知道自己所使用的配置，因此并不需要这两个选项（会使得内核变大）。然而，如果读者要调试的内核是由其他人建立的，则上述选项会比较有用。

CONFIG_ACPI_DEBUG

该选项出现在“Power management/ACPI（电源管理/ACPI）”菜单中。该选项将打开ACPI（Advanced Configuration and Power Interface，高级配置和电源接口）中的详细调试信息。如果怀疑自己所遇到的问题和ACPI相关，则可使用该选项。

CONFIG_DEBUG_DRIVER

在“Device drivers（设备驱动程序）”菜单中。该选项打开驱动程序核心中的调试信息，它可以帮助跟踪底层支持代码中的问题。本书第十四章将阐述驱动程序核心相关的内容。

CONFIG_SCSI_CONSTANTS

该选项出现在“Device drivers/SCSI device support（设备驱动程序/SCSI设备支持）”菜单中，它将打开详细的SCSI错误消息。如果读者要编写SCSI驱动程序，则可使用该选项。

CONFIG_INPUT_EVBUG

该选项可在“Device drivers/Input device support（设备驱动程序/输入设备支持）”中找到，它会打开对输入事件的详细记录。如果读者要针对输入设备编写驱动程序，则可使用该选项。注意该选项会导致的安全问题：它会记录你键入的任何东西，包括密码。

CONFIG_PROFILING

该选项可在“Profiling support（剖析支持）”菜单中找到。剖析通常用于系统性能的调节，但对跟踪内核挂起及相关问题也会有帮助。

在我们讲解不同的内核问题跟踪方法时，将再次遇到上述选项。在此之前，先描述一下经典的调试技术：print语句。

通过打印调试

最普通的调试技术就是监视，即在应用程序编程中，在一些适当的地点调用*printf*显示监视信息。调试内核代码的时候，可以用*printk*来完成相同的工作。

printk

在前面的章节中，我们只是简单假设*printk*工作起来和*printf*很类似。接下来将介绍它们之间的一些不同点。

差别之一就是，通过附加不同日志级别（loglevel），或者说消息优先级，可让*printk*根据这些级别所表示的严重程度对消息进行分类。我们通常采用宏来指示日志级别，例如，KERN_INFO，我们在前面已经看到它被添加在一些打印语句的前面，它就是一个可以使用的消息日志级别。表示日志级别的宏会展开为一个字符串，在编译时由预处理器将它和消息文本拼接在一起；这也就是为什么下面的例子中优先级和格式字串间没有逗号的原因。下面有两个*printk*的例子，一个是调试信息，一个是临界信息：

```
printk(KERN_DEBUG "Here I am: %s:%i\n", __FILE__, __LINE__);
printk(KERN_CRIT "I'm trashed; giving up on %p\n", ptr);
```

在头文件*<linux/kernel.h>*中定义了八种可用的日志级别字符串，下面以严重程度的降序来列出这些级别：

KERN_EMERG
: 用于紧急事件消息，它们一般是系统崩溃之前提示的消息。

KERN_ALERT
: 用于需要立即采取动作的情况。

KERN_CRIT
: 临界状态，通常涉及严重的硬件或软件操作失败。

KERN_ERR
: 用于报告错误状态。设备驱动程序会经常使用KERN_ERR来报告来自硬件的问题。

KERN_WARNING
: 对可能出现问题的情况进行警告，但这类情况通常不会对系统造成严重问题。

KERN_NOTICE
: 有必要进行提示的正常情形。许多与安全相关的状况用这个级别进行汇报。

KERN_INFO
: 提示性信息。很多驱动程序在启动的时候以这个级别来打印出它们找到的硬件信息。

KERN_DEBUG
: 用于调试信息。

每个字符串（以宏的形式展开）表示一个尖括号中的整数。整数值的范围0～7，数值越小，优先级就越高。

未指定优先级的*printk*语句采用的默认级别是DEFAULT_MESSAGE_LOGLEVEL，这个宏在*kernel/printk.c*中被指定为一个整数。在2.6.10内核中，DEFAULT_MESSAGE_LOGLEVEL就是KERN_WARNING，但以前的版本取过不同的值。

根据日志级别，内核可能会把消息打印到当前控制台上，这个控制台可以是一个字符模式的终端、一个串口打印机或是一个并口打印机。当优先级小于console_loglevel这个整数变量的值，消息才能显示出来，而且每次输出一行（如果不以*newline*字符结尾，则不会输出）。如果系统同时运行了*klogd*和*syslogd*，则无论console_loglevel为何值，内核消息都将追加到*/var/log/messages*中（否则按照*syslogd*的配置进行处理）。如果*klogd*没有运行，这些消息就不会传递到用户空间，这种情况下，只能查看*/proc/kmsg*文件（使用*dmesg*命令可以轻松做到）。如果使用*klogd*，则应该了解它不会保存连续相同的信息行；它只会保存连续相同的第一行，并在最后打印这一行的重复次数。

变量console_loglevel的初始值是DEFAULT_CONSOLE_LOGLEVEL，而且还可以通过*sys_syslog*系统调用进行修改。调用*klogd*时可以指定*-c*开关项来修改这个变量，*klogd*的手册页对此有详细说明。注意，要修改其当前值，必须先杀掉*klogd*，然后再用新的*-c*选项重新启动它。此外，还可以编写程序来改变控制台的日志级别。读者可以在O'Reilly的FTP站点提供的源文件*misc-progs/setlevel.c*里找到这样的一段程序。新优先级被指定为一个1～8之间的整数值。如果值被设为1，则只有级别为0（KERN_EMERG）的消息才能到达控制台；如果被设为8，则包括调试信息在内的所有消息都能显示出来。

我们也可以通过对文本文件*/proc/sys/kernel/printk*的访问来读取和修改控制台的日志级别。这个文件包含了4个整数值，分别是：当前的日志级别、未明确指定日志级别时的默认消息级别、最小允许的日志级别以及引导时的默认日志级别。向该文件中写入单个整数值，将会把当前日志级别修改为这个值。例如，可以简单地输入下面的命令使所有的内核消息显示到控制台上：

```
# echo 8 > /proc/sys/kernel/printk
```

现在读者应该清楚为什么在*hello.c*例子中使用了KERN_ALERT标记，因为使用这个标记将确保所有消息都能够显示在控制台上。

重定向控制台消息

对于控制台日志策略，Linux允许有某些灵活性：内核可以将消息发送到一个指定的虚

拟控制台（假如控制台是文本屏幕的话）。默认情况下，“控制台”就是当前的虚拟终端。可以在任何一个控制台设备上调用ioctl(TIOCLINUX)来指定接收消息的其他虚拟终端。下面的*setconsole*程序，可选择专门用来接收内核消息的控制台。这个程序必须由超级用户运行，在*misc-progs*目录里可以找到它。

下面是该程序的完整清单。调用该程序时，请附加一个参数指定要接收消息的控制台编号。

```
int main(int argc, char **argv)
{
    char bytes[2] = {11,0}; /* 11是TIOCLINUX的命令编号 */

    if (argc==  =2) bytes[1] = atoi(argv[1]); /* 选定的控制台 */
    else {
        fprintf(stderr, "%s: need a single arg\n",argv[0]); exit(1);
    }
    if (ioctl(STDIN_FILENO, TIOCLINUX, bytes)<0) {     /* 使用stdin */
        fprintf(stderr,"%s: ioctl(stdin, TIOCLINUX): %s\n",
                argv[0], strerror(errno));
        exit(1);
    }
    exit(0);
}
```

*setconsole*使用了特殊的*ioctl*命令：TIOCLINUX，这个命令可以完成一些特定的Linux功能。使用TIOCLINUX时，需要传给它一个指向字节数组的指针参数。数组的第一个字节指定所请求子命令的编号，随后的字节所具有的功能则由这个子命令来决定。在*setconsole*中，使用的子命令是11，后面那个字节（保存在bytes[1]中）则用来标识虚拟控制台。关于TIOCLINUX的完整描述可以在内核源代码中的*drivers/char/tty_io.c*文件中得到。

消息如何被记录

*printk*函数将消息写到一个长度为__LOG_BUF_LEN字节的循环缓冲区中（我们可在配置内核时为__LOG_BUF_LEN指定4 KB ~ 1 MB之间的值）。然后，该函数会唤醒任何正在等待消息的进程，即那些睡眠在*syslog*系统调用上的进程，或者正在读取*/proc/kmsg*的进程。这两个访问日志引擎的接口几乎是等价的，不过请注意，对*/proc/kmsg*进行读操作时，日志缓冲区中被读取的数据就不再保留，而*syslog*系统调用却能通过选项返回日志数据并保留这些数据，以便其他进程也能使用。一般而言，读*/proc*文件要容易些，这也是*klogd*的默认方法。*dmesg*命令可在不刷新缓冲区的情况下获得缓冲区的内容；实际上，该命令将缓冲区的整个内容返回到*stdout*，而无论该缓冲区是否已经被读取。

如果在停止*klogd*之后手工读取内核消息，读者会发现*/proc/kmsg*文件很像一个FIFO，

读取进程会阻塞在该文件上，以便等待更多的数据。显然，如果已经有 *klogd* 或其他进程正在读取同一数据，就不能采用这种方法读取消息，因为这会与这些进程发生竞争。

如果循环缓冲区填满了，*printk* 就绕回缓冲区的开始处填写新的数据，这将覆盖最陈旧的数据，于是日志进程就会丢失最早的数据。但与使用循环缓冲区所带来的好处相比，这个问题可以忽略不计。例如，循环缓冲区可以使系统在没有记录进程的情况下照样运行，同时覆盖那些不会再有人去读的旧数据，从而使内存的浪费减到最少。Linux 消息处理方法的另一个特点是，可以在任何地方调用 *printk*，甚至在中断处理函数里也可以调用，而且对数据量的大小没有限制。而这个方法的唯一缺点就是可能丢失某些数据。

klogd 运行时会读取内核消息并将它们分发到 *syslogd*，*syslogd* 随后查看 */etc/syslog.conf*，找出处理这些数据的方法。*syslogd* 根据功能和优先级对消息进行区分；这两者的可选值均定义在 *<sys/syslog.h>* 中。内核消息由 LOG_KERN 工具记录，并以与 *printk* 中对应的优先级记录（例如，*printk* 中使用的 KERN_ERR 对应于 *syslogd* 中的 LOG_ERR）。如果没有运行 *klogd*，数据将保留在循环缓冲区中，直到某个进程读取它们或缓冲区溢出为止。

如果想避免因为来自驱动程序的大量监视信息而扰乱系统日志，则可以为 *klogd* 指定 *-f* (file)选项，指示 *klogd* 将消息保存到某个特定的文件，或者修改 */etc/syslog.conf* 来满足自己的需求。另一种可能的办法是采取下面的强制措施：杀掉 *klogd*，而将消息详细地打印到空闲的虚拟终端上（注 1），或者在一个未使用的 *xterm* 上执行命令 *cat /proc/kmsg* 来显示消息。

开启及关闭消息

在驱动程序开发的初期阶段，*printk* 对于调试和测试新代码是相当有帮助的。不过，在正式发布驱动程序时，就得删除这些打印语句，或至少禁用它们。不幸的是，你可能会发现这样的情况，即在删除了那些已被认为不再需要的提示消息后，又需要实现一个新的功能（或是有人发现了一个缺陷），这时，又希望至少重新开启一部分消息。这两个问题可以通过几个办法来解决，以便全局地开启或禁止调试消息，并能对个别消息进行开关控制。

下面给出了一个调用 *printk* 的编码方法，它可个别或全局地开关 *printk* 语句；这个技巧是定义一个宏，在需要时，这个宏展开为一个 *printk*（或 *printf*）调用：

- 可以通过在宏名字中删减或增加一个字母来启用或禁用每一条打印语句。

注 1：例如，使用 *setlevel8;setconsole 10* 来设置终端 10，以显示消息。

- 在编译前修改CFLAGS变量，则可以一次禁用所有消息。
- 同样的打印语句可以在内核代码中也可以在用户级代码使用，因此，关于这些额外的调试信息，驱动程序和测试程序可以用同样的方法来进行管理。

下面这些来自头文件*scull.h*的代码片段就实现了这些功能：

```
#undef PDEBUG             /* 取消对PDEBUG的定义，以防重复定义 */
#ifdef SCULL_DEBUG
#  ifdef _ _KERNEL_ _
     /* 表明打开调试，并处于内核空间 */
#    define PDEBUG(fmt, args...) printk( KERN_DEBUG "scull: " fmt, ##
args)
#  else
     /* 这表明处于用户空间 */
#    define PDEBUG(fmt, args...) fprintf(stderr, fmt, ## args)
#  endif
#else
#  define PDEBUG(fmt, args...) /* 调试被关闭：不作任何事情 */
#endif

#undef PDEBUGG
#define PDEBUGG(fmt, args...) /* 不作任何事情，仅仅是个占位符 */
```

是否定义符号PDEBUG取决于是否定义了SCULL_DEBUG，并且，它能根据代码所运行的环境来选择合适的方式显示信息：在内核态时，它使用内核调用*printk*；在用户空间，则使用*libc*调用*fprintf*，并输出到标准错误设备。另一方面，符号PDEBUGG则什么也不做；它可以将打印语句注释掉，而不必把它们完全删除。

为了进一步简化这个过程，可以在makefile中添加下面几行：

```
# Comment/uncomment the following line to disable/enable debugging
DEBUG = y

# Add your debugging flag (or not) to CFLAGS
ifeq ($(DEBUG),y)
  DEBFLAGS = -O -g -DSCULL_DEBUG # "-O" is needed to expand inlines
else
  DEBFLAGS = -O2
endif

CFLAGS += $(DEBFLAGS)
```

本节所给出的宏依赖于*gcc*对ANSIC预处理器的扩展，这种扩展支持了带可变数目参数的宏。对*gcc*的这种依赖并不是什么问题，因为内核对*gcc*特性的依赖更强。此外，Makefile依赖于GNU的*make*版本；基于同样的道理，这种依赖也不是什么问题。

如果读者熟悉C预处理器，可以对上面的定义进行扩展，实现“调试级别”的概念，这

需要定义一组不同的级别，并为每个级别赋一个整数（或位掩码），用以决定各个级别消息的详细程度。

但是，每一个驱动程序都会有自身的功能和监视需求。良好的编程技术在于选择灵活性和效率的最佳折衷点，我们无法预知对读者来说最合适的点在哪里。记住，预处理条件语句（以及代码中的常量表达式）只在编译时执行，所以要再次打开或关闭消息就必须重新编译。另一种方法就是使用C条件语句，它在运行时执行，因此可以在程序运行期间打开或关闭消息。这是个很好的功能，但每次代码执行时系统都要进行额外的处理，甚至在禁用消息后仍然会影响性能，而有时这种性能损失是无法接受的。

在很多情况下，本节提到的这些宏都已被证实是很有用的，仅有的缺点是每次开启和关闭消息显示时都要重新编译模块。

速度限制

有时读者会一不小心利用*printk*产生了上千条消息，从而让日志信息充满控制台，更可能使系统日志文件溢出。如果使用某个慢速控制台设备（比如串口），过高的消息输出速度会导致系统变慢，甚至使系统无法正常响应。当控制台被无休止的数据填充时，其实很难发现系统到底出现了什么问题。因此，我们应该非常小心地管理自己的打印信息，尤其在驱动程序的正式版本中，或者完成初始化之后。通常，正式代码不应该在正常的操作下打印任何信息，而打印出的信息应作为对需要引起注意的异常情形的提示。

另一方面，在我们驱动的设备停止工作时，也许希望产生一条日志信息。但我们要小心，不能夸张处理这种情况，某些不明智的进程会在遇到失败时不停重试，每秒可能产生成千上万次重试；如果我们的驱动程序每次都打印一条“该设备停止工作”这样的消息，则将制造巨量输出，并可能在控制台设备较慢时独占CPU——我们根本无法中断控制台，尤其在串口或者行式打印机上。

在许多情况下，最好的办法是设置一个标志，表示“我已经就此声明过了，”并在该标志被设置时不再打印任何信息。但在某些情况下，仍然有理由偶尔发出一条“该设备仍停止工作”这样的消息。内核为这种情况提供了一个有用的函数：

```
int printk_ratelimit(void);
```

在打印一条可能被重复的信息之前，应调用上面这个函数。如果该函数返回一个非零值，则可以继续并打印我们的消息，否则就应该跳过。这样，典型的调用应如下所示：

```
if (printk_ratelimit())
    printk(KERN_NOTICE "The printer is still on fire\n");
```

*printk_ratelimit*通过跟踪发送到控制台的消息数量工作。如果输出的速度超过一个阈值，*printk_ratelimit* 将返回零，从而避免发送重复消息。

我们可通过修改 */proc/sys/kernel/printk_ratelimit*（在重新打开消息之前应该等待的秒数）以及 */proc/sys/kernel/printk_ratelimit_burst*（在进行速度限制之前可以接受的消息数）来定制 *printk_ratelimit* 的行为。

打印设备编号

有时当从一个驱动程序打印消息时，我们会希望打印与硬件关联的设备编号。打印设备的主设备号和次设备号并不是很难的事情，但出于一致性考虑，内核提供了一对辅助宏（在 *<linux/kdev_t.h>* 中定义）：

```
int print_dev_t(char *buffer, dev_t dev);
char *format_dev_t(char *buffer, dev_t dev);
```

这两个宏均将设备编号打印到给定的缓冲区，其唯一的区别是，*print_dev_t* 返回的是打印的字符数，而 *format_dev_t* 返回的是缓冲区，这样，它的返回值可直接作为调用 *printk* 时的参数使用。当然，我们不能忘记只有在结尾处存在 newline（新行）字符时，*printk* 才将消息刷新到控制台。传入上述宏的缓冲区必须足够保存一个设备编号。因为在未来的内核版本中，使用 64 位设备编号的可能性非常明显，因此，该缓冲区的大小应该至少有 20 字节长。

通过查询调试

上一节讲述了 *printk* 如何工作以及我们应该如何使用它，但还没谈到它的缺点。

由于 *syslogd* 会一直保持对其输出文件的同步刷新，即使我们可以降低 *console_loglevel* 以避免装载控制台设备，但大量使用 *printk* 仍然会显著降低系统性能。从 syslogd 的角度来看，这样的处理是正确的：它试图把每件事情都记录到磁盘上，以在系统万一崩溃时最后的记录信息能反应崩溃前的状况。然而，因处理调试信息而使系统性能减慢是我们所不希望的。这个问题可以通过在 */etc/syslogd.conf* 中日志文件的名字前面加一个减号前缀来解决（注 2）。修改配置文件带来的问题在于，在完成调试之后这些改动将依旧保留；即使在一般的系统操作中，当希望尽快把信息刷新到磁盘时也是如此。如果不愿

注 2：这个减号是个有魔力的标记，可以避免 *syslogd* 在每次出现新信息时都去刷新磁盘文件，详细文档请见 *syslog.conf(5)*，这个手册页很值得一读。

作这种持久性修改的话，另一个选择是运行一个非*klogd*程序（如前面介绍的*cat/proc/kmsg*），但这样并不能为通常的系统操作提供一个合适的环境。

多数情况中，获取相关信息的最好方法是在需要的时候才去查询系统信息，而不是持续不断地产生数据。实际上，每个Unix系统都提供了很多工具用于获取系统信息，如*ps*、*netstat*、*vmstat*、等等。

驱动程序开发人员可以用如下方法对系统进行查询：在*/proc*文件系统中创建文件、使用驱动程序的*ioctl*方法，以及通过*sysfs*导出属性等。*sysfs*的使用需要驱动程序模型的一些背景知识，因此我们将在第十四章中进行详细描述。

使用 /proc 文件系统

*/proc*文件系统是一种特殊的、由软件创建的文件系统，内核使用它向外界导出信息。*/proc*下面的每个文件都绑定于一个内核函数，用户读取其中的文件时，该函数动态地生成文件的“内容”。我们已经见到过这类文件的一些输出情况，例如，*/proc/modules*列出的是当前载入模块的列表。

在Linux系统中对*/proc*的使用很频繁。现代Linux发行版中的很多工具都是通过*/proc*来获取它们需要的信息，例如*ps*、*top*和*uptime*。有些设备驱动程序也通过*/proc*导出信息，而我们自己的驱动程序当然也可以这么做。因为*/proc*文件系统是动态的，所以驱动程序模块可以在任何时候添加或删除其中的入口项。

具有完整特征的*/proc*入口项可以相当复杂；在所有的这些特征当中，有一点要指出的是，这些*/proc*文件不仅可以用于读出数据，也可以用于写入数据。不过，大多数时候，*/proc*入口项是只读文件。本节将只涉及简单的只读情形。如果有兴趣实现更为复杂的事情，读者可以先在这里了解基础知识，然后参考内核源代码来建立完整的认识。

但在继续之前，首先要指出我们并不鼓励在*/proc*下添加文件。相比最初的用途（用于提供系统中进程的信息），*/proc*文件系统已经不受控制地增加了大量信息。因此，我们建议新的代码通过sysfs来向外界导出信息。先前提到，对sysfs的利用需要对Linux设备模型的理解，因此我们将在第十四章讲述。而目前，*/proc*目录下的文件更容易创建，并且完全符合调试用途，因此我们在本节讲述这一方法。

在 /proc 中实现文件

所有使用*/proc*的模块必须包含*<linux/proc_fs.h>*，并通过这个头文件来定义正确的函数。

为创建一个只读的*/proc*文件，驱动程序必须实现一个函数，用于在读取文件时生成数据。当某个进程读取这个文件时（使用*read*系统调用），读取请求会通过这个函数发送到驱动程序模块。我们把注册接口放到本节后面，先直接讲述这个函数。

在某个进程读取我们的*/proc*文件时，内核会分配一个内存页（即PAGE_SIZE字节的内存块），驱动程序可以将数据通过这个内存页返回到用户空间。该缓冲区会传入我们定义的函数，而该函数称为*read_proc*方法：

```
int (*read_proc)(char *page, char **start, off_t offset, int count,
                 int *eof, void *data);
```

参数表中的page指针指向用来写入数据的缓冲区；函数应使用start返回实际的数据写到内存页的哪个位置（对此后面还将进一步谈到）；offset和count这两个参数与*read*方法相同。eof参数指向一个整型数，当没有数据可返回时，驱动程序必须设置这个参数；data参数是提供给驱动程序的专用数据指针，可用于内部记录。

该函数必须返回存放到内存页缓冲区的字节数，这一点与*read*函数对其他类型文件的处理相同。另外还有*eof和*start这两个输出值。eof只是一个简单的标志，而start的用法就有点复杂了，它可以帮助实现大（大于一个内存页）的*/proc*文件。

start参数的用法看起来有些特别，它用来指示要返回给用户的数据保存在内存页的什么位置。在我们的*read_proc*方法被调用时，*start的初始值为NULL。如果保留*start为空，内核将假定数据保存在内存页偏移量0的地方；也就是说，内核将对*read_proc*作如下简单假定：该函数将虚拟文件的整个内容放到了内存页，并同时忽略offset参数。相反，如果我们将*start设置为非空值，内核将认为由*start指向的数据是offset指定的偏移量处的数据，可直接返回给用户。通常，返回少量数据的简单*read_proc*方法可忽略start参数，复杂的*read_proc*方法会将*start设置为页面，并将所请求偏移量处的数据放到内存页中。

长久以来，关于*/proc*文件还有另一个主要问题，这也是start意图解决的一个问题。有时，在连续的*read*调用之间，内核数据结构的ASCII表述会发生变化，以至于读取进程发现前后两次调用所获得的数据不一致。如果把*start设为一个小的整数值，那么调用程序可以利用它来增加filp->f_pos的值，而不依赖于返回的数据量，因此也就使f_pos成为*read_proc*过程的一个内部记录值。例如，如果*read_proc*函数从一个大的结构数组返回数据，并且这些结构的前五个已经在第一次调用中返回，那么可将*start设置为5。下次调用中这个值将被作为偏移量；驱动程序也就知道应该从数组的第六个结构开始返回数据。这种方法被它的作者称作“hack”，可以在*/fs/proc/generic.c*中看到。

注意，还有一个更好的方法可实现*/proc*文件，该方法称为seq_file，我们稍后将讲述这个方法。现在我们来看一个例子，下面是*scull*设备*read_proc*函数的简单实现：

```
int scull_read_procmem(char *buf, char **start, off_t offset,
                   int count, int *eof, void *data)
{
    int i, j, len = 0;
    int limit = count - 80; /* 不要打印超过这个值的数据 */

    for (i = 0; i < scull_nr_devs && len <= limit; i++) {
        struct scull_dev *d = &scull_devices[i];
        struct scull_qset *qs = d->data;
        if (down_interruptible(&d->sem))
            return -ERESTARTSYS;
        len += sprintf(buf+len,"\nDevice %i: qset %i, q %i, sz %li\n",
                i, d->qset, d->quantum, d->size);
        for (; qs && len <= limit; qs = qs->next) { /* scan the list */
            len += sprintf(buf + len, "  item at %p, qset at %p\n",
                    qs, qs->data);
            if (qs->data && !qs->next) /* 只转储最后一项 */
                for (j = 0; j < d->qset; j++) {
                    if (qs->data[j])
                        len += sprintf(buf + len,
                                "    % 4i: %8p\n",
                                j, qs->data[j]);
                }
        }
        up(&scull_devices[i].sem);
    }
    *eof = 1;
    return len;
}
```

这是一个相当典型的*read_proc*实现。它假定决不会有这样的需求，即生成多于一页的数据，因此忽略了start和offset值。但是，小心不要超出缓冲区，以防万一。

老的/proc接口

如果读者通读内核源代码，会发现某些*/proc*文件通过下面的老接口实现：

```
int (*get_info)(char *page, char **start, off_t offset, int count);
```

其中所有的参数都和*read_proc*方法一样，只是没有eof和data参数。该接口仍然被支持，但可能在将来被取消；因此，新的代码应该使用*read_proc*接口。

创建自己的/proc文件

一旦定义好了一个*read_proc*函数，就需要把它与一个*/proc*入口项连接起来。这通过调用*create_proc_read_entry*实现：

```
struct proc_dir_entry *create_proc_read_entry(const char *name,
                                mode_t mode, struct proc_dir_entry *base,
                                read_proc_t *read_proc, void *data);
```

其中，name是要创建的文件名称；mode是该文件的保护掩码（可传入0表示系统默认值）；base指定该文件所在的目录（如果base为NULL，则该文件将创建在*/proc*的根目录）；*read_proc*是实现该文件的*read_proc*函数；内核会忽略data参数，但是会将该参数传递给*read_proc*。下面是*scull*调用该函数创建*/proc*文件的代码：

```
create_proc_read_entry("scullmem", 0 /* default mode */,
        NULL /* parent dir */, scull_read_procmem,
        NULL /* client data */);
```

上述代码在*/proc*目录下创建了一个称为*scullmem*的文件，并默认具有全局可读权限设置。

目录项指针可用来在*/proc*下创建完整的目录层次结构。不过请注意，将入口项置于*/proc*的子目录中有更为简单的方法，即把目录名称作为入口项名称的一部分 —— 只要目录本身已经存在。例如，有个经常被忽略的约定，要求把设备驱动程序对应的*/proc*入口项转移到子目录*driver/*中。*scull*可以简单地指定它的入口项名称为*driver/scullmem*，从而把它的*/proc*文件放到这个子目录中。

当然，在卸载模块时，*/proc*中的入口项也应被删除。*remove_proc_entry*就是用来撤销*create_proc_read_entry*所做工作的函数：

```
remove_proc_entry("scullmem", NULL /* parent dir */);
```

如果删除入口项失败，将导致未预期的调用，如果模块已被卸载，内核会崩溃。

在使用*/proc*文件时，读者必须谨记这种实现的几个不足之处 —— 因此我们不鼓励使用*/proc*文件。

最重要的问题和*/proc*项的删除有关。删除调用可能在文件正在被使用时发生，因为*/proc*入口项不存在关联的所有者，因此对这些文件的使用并不会作用到模块的引用计数上。在移除模块时，执行*sleep 100 < /proc/myfile*命令就可以触发这个问题。

另外一个问题是关于使用同一名字注册两个入口项。内核信任驱动程序，因此不会检查某个名称是否已经被注册，因此如果不小心，将可能导致两个或多个入口项具有相同的名字。这种“重名”问题经常会出现在“教室”中。由于入口项无法区分，因此不论是访问这些入口项的时候还是调用*remove_proc_entry*的时候，都会出现问题。

seq_file 接口

上面提到，*/proc* 下大文件的实现有些笨拙。随着时间的流逝，当越来越多的开发者使用 */proc* 文件输出信息时，*/proc* 的实现接口变得越来越“声名狼藉”。为了让内核开发工作更加容易，通过对 */proc* 代码的整理而增加了 seq_file 接口。这一接口为大的内核虚拟文件提供了一组简单的函数。

seq_file接口假定我们正在创建的虚拟文件要顺序遍历一个项目序列，而这些项目正是必须要返回给用户空间的。为使用 seq_file，我们必须创建一个简单的“迭代器(iterator)”对象，该对象用来表示项目序列中的位置，每前进一步，该对象输出序列中的一个项目。这听起来有些复杂，但实际上整个过程相当简单。下面我们将使用这个方法针对 *scull* 驱动程序创建一个 */proc* 文件。

显然，第一步是包含 *<linux/seq_file.h>* 头文件，然后必须建立四个迭代器对象，分别为 *start*、*next*、*stop* 和 *show*。

start 方法始终会首先调用，该函数的原型如下：

```
void *start(struct seq_file *sfile, loff_t *pos);
```

这里的 sfile 参数几乎可在大多数情况下忽略。pos 是一个整数的位置值，表明读取的位置。对位置的解释完全取决于迭代器的实现本身，并不一定非得是结果文件的字节位置。因为seq_file的实现通常都要遍历一个项目序列，因此位置通常被解释为指向序列中下一个项目的游标(cursor)。*scull* 驱动程序将每个设备当作序列中的一个项目，这样，传入的pos就可以简单作为scull_devices数组的索引。于是，*scull* 的 *start* 方法可如下编写：

```
static void *scull_seq_start(struct seq_file *s, loff_t *pos)
{
    if (*pos >= scull_nr_devs)
        return NULL;    /* 无数据可返回 */
    return scull_devices + *pos;
}
```

如果返回值非NULL，则迭代器的实现可将其作为私有值使用。

next 函数应将迭代器移动到下一个位置，并在序列中没有其他项目时返回 NULL。该方法的原型是：

```
void *next(struct seq_file *sfile, void *v, loff_t *pos);
```

其中，v是先前对 *start* 或者 *next* 的调用所返回的迭代器，pos 是文件的当前位置。*next* 方法应增加pos指向的值，这依赖于迭代器的工作方式，在某些情况下，我们也许要让 pos 的增加值大于 1。*scull* 的 *next* 方法如下实现：

```
static void *scull_seq_next(struct seq_file *s, void *v, loff_t *pos)
{
    (*pos)++;
    if (*pos >= scull_nr_devs)
        return NULL;
    return scull_devices + *pos;
}
```

当内核使用迭代器之后，会调用 *stop* 方法通知我们进行清除工作：

```
void stop(struct seq_file *sfile, void *v);
```

scull 的实现不需要完成清除工作，因此它的 *stop* 方法为空。

值得注意的是，在设计上，seq_file的代码不会在 *start* 和 *stop* 的调用之间执行其他的非原子操作。我们可以确信，*start* 被调用之后马上就会有对 *stop* 的调用。因此，在 *start* 方法中获取信号量或者自旋锁是安全的。只要其他seq_file方法是原子的，则整个调用过程也是原子的（如果对这段描述理解起来有些困难，读者可在阅读下一章之后再回来阅读）。

在上述调用之间，内核会调用 *show* 方法来将实际的数据输出到用户空间。该方法的原型如下：

```
int show(struct seq_file *sfile, void *v);
```

该方法应该为迭代器 v 所指向的项目建立输出。但是，它不能使用 *printk* 函数，而要使用针对 seq_file 输出的一组特殊函数：

`int seq_printf(struct seq_file *sfile, const char *fmt, ...);`

这是seq_file实现的 *printf* 等价函数；它需要通常的格式字符串以及额外的值参数。同时，我们还要将 *show* 函数传入的 seq_file 结构传递给这个函数。如果 *seq_printf* 返回了一个非零值，则意味着缓冲区已满，而输出被丢弃。大部分实现都会忽略这个返回值。

`int seq_putc(struct seq_file *sfile, char c);`

`int seq_puts(struct seq_file *sfile, const char *s);`

这两个函数是用户空间常用的 *putc* 和 *puts* 函数的等价函数。

`int seq_escape(struct seq_file *m, const char *s, const char *esc);`

这个函数等价于 *seq_puts*，只是若 s 中的某个字符也存在于esc中，则该字符会以八进制形式打印。传递给 esc 参数的常见值是 "\t\n\\"，它可以避免要输出的空白字符弄乱屏幕或者迷惑 shell 脚本。

```
int seq_path(struct seq_file *sfile, struct vfsmount *m, struct dentry
  *dentry, char *esc);
```

这个函数可用于输出与某个目录项关联的文件名。对设备驱动程序来讲，它没有多少价值，这里包含该函数只是出于完整性考虑。

在我们的例子中，*scull* 中使用的 *show* 方法代码如下所示：

```
static int scull_seq_show(struct seq_file *s, void *v)
{
    struct scull_dev *dev = (struct scull_dev *) v;
    struct scull_qset *d;
    int i;

    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
    seq_printf(s, "\nDevice %i: qset %i, q %i, sz %li\n",
            (int) (dev - scull_devices), dev->qset,
            dev->quantum, dev->size);
    for (d = dev->data; d; d = d->next) { /* scan the list */
        seq_printf(s, "  item at %p, qset at %p\n", d, d->data);
        if (d->data && !d->next) /* 只转储最后一项 */
            for (i = 0; i < dev->qset; i++) {
                if (d->data[i])
                    seq_printf(s, "    % 4i: %8p\n",
                            i, d->data[i]);
            }
    }
    up(&dev->sem);
    return 0;
}
```

这里，我们最终解释了自己的“迭代器”值，它实际就是一个指向 scull_dev 结构的指针。

现在，我们定义了完整的迭代器操作函数，*scull* 必须将这些函数打包并和 */proc* 中的某个文件连接起来。首先要填充一个 seq_operations 结构：

```
static struct seq_operations scull_seq_ops = {
    .start = scull_seq_start,
    .next  = scull_seq_next,
    .stop  = scull_seq_stop,
    .show  = scull_seq_show
};
```

有了这个结构，我们必须创建一个内核能够理解的文件实现。在使用 seq_file 时，我们不使用先前描述过的 *read_proc* 方法，而最好在略低的层次上连接到 */proc*。也就是说，我们将创建一个 file_operations 结构（即用于字符驱动程序的相同结构），这个结构

将实现内核在该*/proc*文件上进行读取和定位时所需的所有操作。幸运的是，这一过程非常直接。首先创建一个*open*方法，该方法将文件连接到seq_file操作：

```
static int scull_proc_open(struct inode *inode, struct file *file)
{
    return seq_open(file, &scull_seq_ops);
}
```

对*seq_open*的调用将file结构和我们上面定义的顺序操作连接在一起。*open*是唯一一个必须由我们自己实现的文件操作，因此，我们的file_operations结构可如下定义：

```
static struct file_operations scull_proc_ops = {
    .owner   = THIS_MODULE,
    .open    = scull_proc_open,
    .read    = seq_read,
    .llseek  = seq_lseek,
    .release = seq_release
};
```

这里，我们指定了我们自己的*open*方法，但对其他的*file_operations*成员，我们使用了已经定义好的*seq_read*、*seq_lseek*和*seq_release*方法。

最后，我们建立实际的*/proc*文件：

```
entry = create_proc_entry("scullseq", 0, NULL);
if (entry)
    entry->proc_fops = &scull_proc_ops;
```

这次，我们没有使用*create_proc_read_entry*函数，而是使用了低层的*create_proc_entry*，它的原型定义如下：

```
struct proc_dir_entry *create_proc_entry(const char *name,
                                         mode_t mode,
                                         struct proc_dir_entry *parent);
```

该函数的参数和*create_proc_read_entry*等价，分别是文件的名称（name）、访问保护掩码（mode）以及父（parent）目录。

利用上面的代码，*scull*就在*/proc*中拥有了一个和先前版本类似的文件。但显然，这个文件要更加灵活一些，不管输出有多大，它都能够正确处理文件定位，并且相关代码更加容易阅读和维护。如果读者的*/proc*文件包含有大量的输出行，则我们建议使用seq_file接口来实现该文件。

ioctl方法

*ioctl*是作用于文件描述符之上的一个系统调用，我们会在第六章介绍它的用法。*ioctl*接

收一个“命令”号以及另一个（可选的）参数，命令号用以标识将要执行的命令，而可选参数通常是个指针。作为替代/*proc*文件系统的方法，我们可以专为调试设计若干*ioctl*命令。这些命令从驱动程序复制相关的数据到用户空间，然后可在用户空间中检验这些数据。

使用*ioctl*获取信息比起*/proc*来要困难一些，因为需要另一个程序调用*ioctl*并显示结果。我们必须编写并编译这个程序，还要和需要测试的模块保持同步。但从另一方面来说，相对实现*/proc*文件所需的工作，驱动程序端的编码则更为容易些。

有时*ioctl*是获取信息的最好方法，因为它比起读*/proc*要快得多。如果在数据写到屏幕之前要完成某些处理工作，那么以二进制获取数据要比读取文本文件有效得多。此外，*ioctl*并不要求把数据分割成不超过一个内存页面的片断。

*ioctl*方法的另一个有意思的优点是，甚至在调试被禁用之后，用来取得信息的这些命令仍可以保留在驱动程序中。*/proc*文件对任何查看这个目录的人都是可见的（很多人可能会纳闷“这些奇怪的文件是用来做什么的”），然而与*/proc*文件不同，未公开的*ioctl*命令通常都不会被注意到。此外，万一驱动程序有什么异常，这些命令仍然可以用来调试。唯一的缺点就是模块会稍微大一些。

通过监视调试

有时，通过监视用户空间中应用程序的运行情况，可以捕捉到一些小问题。监视程序同样也有助于确认驱动程序工作是否正常。例如，查看*scull*的*read*实现如何响应不同数据量的*read*请求，就可以判断它是否工作正常。

有许多方法可用来监视用户空间程序的工作情况，比如用调试器一步步跟踪它的函数，插入打印语句，或者在*strace*状态下运行程序等等。在检查内核代码时，最后一项技术最值得关注，我们将在此对它进行讨论。

*strace*命令是一个功能非常强大的工具，它可以显示由用户空间程序所发出的所有系统调用。它不仅可以显示调用，而且还能显示调用参数以及用符号形式表示的返回值。当系统调用失败时，错误的符号值（如ENOMEM）和对应的字符串（如“`Out of memory`，内存溢出”）都能被显示出来。*strace*有许多命令行选项，其中最为有用的是下面几个：*-t*，该选项用来显示调用发生的时间；*-T*，显示调用所花费的时间；*-e*，限定被跟踪的调用类型；*-o*，将输出重定向到一个文件中。默认情况下，*strace*将跟踪信息打印到`stderr`上。

*strace*从内核中接收信息。这意味着一个程序无论是否以支持调试的方式编译（用*gcc*

的-g选项）或是否被去掉了符号信息，都可以被跟踪。调试器可以连接到一个正在运行的进程并控制该进程，而 *strace* 也可以跟踪一个正在运行的进程。

跟踪信息通常用于生成错误报告，然后把它们发送给应用程序开发人员，但是它对内核编程人员来说也同样非常有用。我们已经看到驱动程序代码是如何通过响应系统调用而得到执行的，*strace* 允许我们检查每次调用中输入和输出数据的一致性。

例如，下面给出了 *strace ls /dev > /dev/scull0* 命令的最后几行输出信息：

```
open("/dev", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY) = 3
fstat64(3, {st_mode=S_IFDIR|0755, st_size=24576, ...}) = 0
fcntl64(3, F_SETFD, FD_CLOEXEC)          = 0
getdents64(3, /* 141 entries */, 4096)   = 4088
[...]
getdents64(3, /* 0 entries */, 4096)     = 0
close(3)                                 = 0
[...]
fstat64(1, {st_mode=S_IFCHR|0664, st_rdev=makedev(254, 0), ...}) = 0
write(1, "MAKEDEV\nadmmidi0\nadmmidi1\nadmmid"..., 4096) = 4000
write(1, "b\nptywc\nptywd\nptywe\nptywf\nptyx0\n"..., 96) = 96
write(1, "b\nptyxc\nptyxd\nptyxe\nptyxf\nptyy0\n"..., 4096) = 3904
write(1, "s17\nvcs18\nvcs19\nvcs2\nvcs20\nvcs21"..., 192) = 192
write(1, "\nvcs47\nvcs48\nvcs49\nvcs5\nvcs50\nvc"..., 673) = 673
close(1)                                 = 0
exit_group(0)                            = ?
```

很明显，当 *ls* 完成对目标目录的检索后，在首次对 *write* 的调用中，它试图写入4KB数据。很奇怪的是（对于 *ls* 来说），实际只写入了4000个字节，接着它重试这一操作。然而，我们知道 *scull* 的 *write* 实现每次最多只写入一个量子（*scull* 中设置的量子大小为4000个字节），所以我们所预期的就是上述的部分写入。经过几个步骤之后，每件工作都顺利通过，程序正常退出。

下面是另一个例子，让我们来对 *scull* 设备进行读操作（使用 wc 命令）：

```
[...]
open("/dev/scull0", O_RDONLY|O_LARGEFILE) = 3
fstat64(3, {st_mode=S_IFCHR|0664, st_rdev=makedev(254, 0), ...}) = 0
read(3, "MAKEDEV\nadmmidi0\nadmmidi1\nadmmid"..., 16384) = 4000
read(3, "b\nptywc\nptywd\nptywe\nptywf\nptyx0\n"..., 16384) = 4000
read(3, "s17\nvcs18\nvcs19\nvcs2\nvcs20\nvcs21"..., 16384) = 865
read(3, "", 16384)                       = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0
write(1, "8865 /dev/scull0\n", 17)       = 17
close(3)                                 = 0
exit_group(0)                            = ?
```

正如我们所料，*read* 每次只能读取4000个字节，但数据总量与前面例子中写入的总量是相同的。与上面的写入跟踪相比，请注意本例中的重试是如何组织的。为了快速读取

数据，*wc*已被优化了，因而它绕过了标准库，试图通过一次系统调用读取更多的数据，这可以从`read`的跟踪行中看到：*wc*每次均试图读取16KB数据。

Linux专家可以在*strace*的输出中发现很多有用信息，但如果觉得这些符号过于拖累的话，则可以仅限于监视文件方法（*open*、*read*等）的工作过程。

就个人观点而言，笔者发现*strace*对于查找系统调用运行时的细微错误最为有用。通常应用程序或演示程序中的*perror*调用信息在用于调试时还不够详细，而*strace*能够确切查明系统调用的哪个参数引发了错误，这一点对调试是大有帮助的。

调试系统故障

即使采用了所有这些监视和调试技术，有时驱动程序中依然会有错误，这样的驱动程序在执行时就会产生系统故障。在出现这种情况时，获取尽可能多的信息对解决问题是至关重要的。

注意，“故障（fault）”并不意味着“惊恐（panic）”。Linux代码非常健壮，可以很好地响应大部分错误：故障通常会导致当前进程崩溃，而系统仍会继续运行。如果在进程上下文之外发生了故障，或是系统的关键部分被损害时，系统才有可能panic。但如果问题出现在驱动程序中，通常只会导致正在使用驱动程序的那个进程突然终止。唯一不可恢复的损失就是，当进程被终止时为进程上下文分配的一些内存可能会丢失；例如，驱动程序通过*kmalloc*分配的动态链表可能丢失。然而，由于内核在进程终止时会对已打开的设备调用进行*close*操作，驱动程序仍可以释放由*open*方法分配的资源。

尽管oops消息通常并不会导致整个系统崩溃，但我们发现遇到此类情况时还是要重新引导系统。一个有缺陷的驱动程序可能导致硬件不可用，或者导致内核资源处于不一致的状态，或者在最坏的情况下随机破坏内核内存。通常，我们可以在看到oops之后卸载自己有缺陷的驱动程序，然后重试。但是，如果我们看到任何说明系统整体出现问题的信息后，最好的办法就是立即重新引导系统。

我们已经说过，当内核行为异常时，会在控制台上打印出提示信息。下一节将说明如何解码并使用这些消息。尽管它们对于初学者来说相当晦涩难懂，不过处理器在出错时转储出的这些数据包含了许多值得关注的信息，通过它们通常足以查明程序错误，而无需额外的测试。

oops消息

大部分错误都是因为对NULL指针取值或因为使用了其他不正确的指针值。这些错误通常会导致一个oops消息。

由处理器使用的地址几乎都是虚拟地址，这些地址（除了内存管理子系统本身所使用的物理内存之外）通过一个复杂的被称为“页表”的结构被映射为物理地址。当引用一个非法指针时，分页机制无法将该地址映射到物理地址，此时处理器就会向操作系统发出一个“页面失效（page fault）”的信号。如果地址非法，内核就无法“换入（page in）”缺失页面；这时，如果处理器恰好处于超级用户模式，系统就会产生一个oops。

oops显示发生错误时处理器的状态，比如CPU寄存器的内容以及其他看上去无法理解的信息。这些消息由失效处理函数（*arch/*/kernel/traps.c*）中的*printk*语句产生，就像前面“printk”一节所介绍的那样处理。

让我们看看oops消息的例子。当我们在一台运行2.6版内核的PC机上使用一个NULL指针时，就会导致下面这些信息被显示出来。这里最为相关的信息就是指令指针（EIP），即出错指令的地址。

```
Unable to handle kernel NULL pointer dereference at virtual address 00000000
 printing eip:
d083a064
Oops: 0002 [#1]
SMP
CPU:    0
EIP:    0060:[<d083a064>]    Not tainted
EFLAGS: 00010246   (2.6.6)
EIP is at faulty_write+0x4/0x10 [faulty]
eax: 00000000   ebx: 00000000   ecx: 00000000   edx: 00000000
esi: cf8b2460   edi: cf8b2480   ebp: 00000005   esp: c31c5f74
ds: 007b   es: 007b   ss: 0068
Process bash (pid: 2086, threadinfo=c31c4000 task=cfa0a6c0)
Stack: c0150558 cf8b2460 080e9408 00000005 cf8b2480 00000000 cf8b2460 cf8b2460
       fffffff7 080e9408 c31c4000 c0150682 cf8b2460 080e9408 00000005 cf8b2480
       00000000 00000001 00000005 c0103f8f 00000001 080e9408 00000005 00000005
Call Trace:
 [<c0150558>] vfs_write+0xb8/0x130
 [<c0150682>] sys_write+0x42/0x70
 [<c0103f8f>] syscall_call+0x7/0xb

Code: 89 15 00 00 00 00 c3 90 8d 74 26 00 83 ec 0c b8 00 a6 83 d0
```

这个消息是通过对*faulty*模块的一个设备进行写操作而产生的，*faulty*模块专为演示出错而编写。*faulty.c*中*write*方法的实现很简单：

```
ssize_t faulty_write (struct file *filp, const char __user *buf, size_t count,
        loff_t *pos)
{
    /* make a simple fault by dereferencing a NULL pointer */
    *(int *)0 = 0;
    return 0;
}
```

正如读者所见，我们在这里引用了一个NULL指针。因为0决不会是个合法的指针值，所以产生了错误，内核进入上面的oops消息状态。这个调用进程接着就被杀掉了。

在*faulty*模块的*read*实现中，该模块还展示了更多有意思的错误状态：

```
ssize_t faulty_read(struct file *filp, char _ _user *buf,
            size_t count, loff_t *pos)
{
    int ret;
    char stack_buf[4];

    /* 试着产生缓冲区溢出错误 */
    memset(stack_buf, 0xff, 20);
    if (count > 4)
        count = 4; /* 为用户空间复制四个字节 */
    ret = copy_to_user(buf, stack_buf, count);
    if (!ret)
        return count;
    return ret;
}
```

该方法将一个字符串复制到一个局部变量，但不幸的是，字符串要比目标数组长。这样就会在该函数返回时因为缓冲区溢出而导致一个oops的产生。然而，由于return指令把指令指针带到了无法预期的地方，所以这种错误很难跟踪，所能获得的仅是如下的信息：

```
EIP:    0010:[<00000000>]
Unable to handle kernel paging request at virtual address ffffffff
 printing eip:
ffffffff
Oops: 0000 [#5]
SMP
CPU:    0
EIP:    0060:[<ffffffff>]    Not tainted
EFLAGS: 00010296   (2.6.6)
EIP is at 0xffffffff
eax: 0000000c   ebx: ffffffff   ecx: 00000000   edx: bfffda7c
esi: cf434f00   edi: ffffffff   ebp: 00002000   esp: c27fff78
ds: 007b   es: 007b   ss: 0068
Process head (pid: 2331, threadinfo=c27fe000 task=c3226150)
Stack: ffffffff bfffda70 00002000 cf434f20 00000001 00000286 cf434f00 ffffffff
       bfffda70 c27fe000 c0150612 cf434f00 bfffda70 00002000 cf434f20 00000000
       00000003 00002000 c0103f8f 00000003 bfffda70 00002000 00002000 bfffda70
Call Trace:
 [<c0150612>] sys_read+0x42/0x70
 [<c0103f8f>] syscall_call+0x7/0xb

Code:  Bad EIP value.
```

在这种情况下，我们只能看到调用栈的部分信息（无法看到*vfs_read*和*faulty_read*），内

核抱怨说遇到一条“错误的EIP值（bad EIP value）”。这一抱怨，以及开头处列出的明显错误的地址（ffffffff）均说明内核栈已经被破坏。

通常，在我们面对一条oops时，首先要观察的是发生的问题所在的位置，这通常可通过调用栈信息得到。在上面给出的第一个oops中，相关的信息是：

```
EIP is at faulty_write+0x4/0x10 [faulty]
```

从这里我们可以看到，故障所在的函数是*faulty_write*，该函数位于*faulty*模块（列在中括号内）。十六进制的数据表明指令指针在该函数的4字节处，而函数本身是10（十六进制）字节长。通常，这些信息足以让我们看到问题的真正所在。

如果需要更多信息，调用栈可以告诉我们系统是如何到达故障点的。栈本身以十六进制形式打印，通过一些工作，我们可通过栈清单确定局部变量和函数参数的值。有经验的内核开发人员通过此类模式可有效地发现问题所在。例如，如果我们观察*faulty_read*产生的oops的栈清单：

```
Stack: ffffffff bfffda70 00002000 cf434f20 00000001 00000286 cf434f00 ffffffff7
       bfffda70 c27fe000 c0150612 cf434f00 bfffda70 00002000 cf434f20 00000000
       00000003 00002000 c0103f8f 00000003 bfffda70 00002000 00002000 bfffda70
```

栈顶部的ffffffff就是导致故障产生的字符串的一部分。在x86架构上，用户空间的栈默认自0xc0000000向下。因此，很容易联想到0xbfffda70可能是用户空间的栈地址，亦即传递给*read*系统调用的缓冲区地址，这个地址会在内核的调用链上重复向下传递。在x86架构上（仍然是默认情况下），内核空间起始于0xc0000000，故大于0xc0000000的值几乎肯定是内核空间的地址，等等。

最后，在观察oops清单时还要记得观察本章前面讨论过的“slab毒剂”值。例如，如果我们获得的内核oops中包含有0xa5a5a5a5这样的地址，那几乎可以肯定的是，我们在某处忘记了初始化动态分配到的内存。

需要注意的是，只有在构造内核时打开了CONFIG_KALLSYMS选项，我们才能看到符号化的调用栈（就像上面列出的那样）；否则，我们只能看到裸的、十六进制的清单，因而只有通过其他途径解开这些数字的含义，才能弄清楚真正的调用栈。

系统挂起

尽管内核代码中的大多数错误只会导致一个oops消息，但有时它们会将系统完全挂起。如果系统挂起了，任何消息都无法打印出来。例如，如果代码进入一个死循环，内核就

会停止调度（注3），系统不会再响应任何动作，包括Ctrl-Alt-Del组合键。处理系统挂起有两个选择——要么是防患于未然，要么是亡羊补牢，在发生挂起后调试代码。

通过在一些关键点上插入*schedule*调用可以防止死循环。*schedule*函数（正如读者猜到的）会调用调度器，并因此允许其他进程“偷取”当前进程的CPU时间。如果该进程因驱动程序的错误而在内核空间陷入死循环，则可以在跟踪到这种情况之后，借助*schedule*调用杀死这个进程。

当然，应该意识到任何对*schedule*的调用都可能给驱动程序带来代码重入的问题，因为*schedule*允许其他进程开始运行。如果我们在驱动程序中使用了合适的锁定，这种重入通常不会带来问题。不过，一定不要在驱动程序持有自旋锁的任何时候调用*schedule*。

如果驱动程序确实会挂起系统，而你又不知该在什么位置插入*schedule*调用时，最好的方法是加入一些打印信息，并把它们写入控制台（必要时修改`console_loglevel`的值）。

有时系统看起来像挂起了，但其实并没有。例如，如果键盘因某种奇怪的原因被锁住了就会发生这种情况。这时，运行专为探明此种情况而设计的程序，通过查看它的输出情况，可以发现这种假的挂起。显示器上的时钟或系统负荷表就是很好的状态监视器；只要这些程序保持更新，就说明调度器仍在工作。

对于上述情形，一个不可缺少的工具是“SysRq魔法键（magic SysRq key）”，大多数架构上都可以利用魔法键。SysRq魔法可通过PC键盘上的ALT和SysRq组合键来激活，在其他平台上则通过其他特殊键激活（详情可见*Documentation/sysrq.txt*），串口控制台上也可激活。根据与这两个键一起按下的第三个键的不同，内核会执行许多有用动作中的其中一个，如下所示：

`r` 关闭键盘的raw模式。当某个崩溃的应用程序（比如X服务器）让键盘处于一种奇怪状态时，就可以用这个键关闭raw模式。

`k` 激活“留意安全键（secure attention key，SAK）”功能。SAK将杀死当前控制台上运行的所有进程，留下一个干净的终端。

`s` 对所有磁盘进行紧急同步。

`u` 尝试以只读模式重新挂装所有磁盘。这个操作通常紧接着*s*动作之后立即被调用，它可以在系统处于严重故障状态时节省很多检查文件系统的时间。

注3：实际上，多处理器系统仍然会在其他处理器上调度，即使是单处理器系统，如果内核是可抢占的，也会重新调度。但在大多数常见情形（禁止抢占的单处理器）下，系统会整个停止调度。

b　立即重启系统。注意先要执行同步并重新挂装磁盘。

p　打印当前的处理器寄存器信息。

t　打印当前的任务列表。

m　打印内存信息。

其他一些SysRq功能的信息可参阅内核源代码*Documentation*目录下的*sysrq.txt*文件。注意，SysRq功能必须显式地在内核配置中启用，出于安全原因，大多数发行版并未启用这一功能。不过，对于一个用于驱动程序开发的系统来说，为启用SysRq功能而带来的重新编译新内核的麻烦是值得的。在系统运行时，可通过下面的命令启用SysRq功能：

```
echo 0 > /proc/sys/kernel/sysrq
```

如果未授权用户可以使用系统的键盘，则应该考虑禁止这个功能，以避免出现意外或者蓄意的破坏。以前的一些内核版本默认禁止SysRq功能，因此，在运行时可向上面的*/proc/sys*文件中写入1来打开该功能。

因为SysRq功能非常有用，因此这些功能也对无法访问控制台的系统管理员开放。*/proc/sysrq-trigger*是一个只写的*/proc*入口点，向这个文件写入对应的字符，就可以触发相应的SysRq动作。这个针对SysRq的入口点始终可用，即使控制台上的SysRq是禁止的。

如果读者遇到“活的挂起”，即驱动程序进入了某个死循环但系统整体还能工作，则值得了解针对这种情况的一些技巧。通常，SysRq的*p*功能可直接指出有问题的例程所在的位置。如果这种方法行不通，还可以使用内核剖析功能。构造一个打开剖析功能的内核，并通过引导命令行参数`profile=2`引导该内核。利用*readprofile*工具重置剖析计数器，然后让驱动程序进入死循环状态。经过一段时间之后，再次使用*readprofile*即可观察到浪费CPU资源的内核位置。另外一个更加高级的方法是使用*oprofile*。内核源代码中的*Documentation/basic_profiling.txt*文件详细描述了剖析器相关的所有东西。

在复现系统的挂起故障时，另一个要采取的预防措施是，把所有的磁盘以只读的方式挂装在系统上（或干脆卸装它们）。如果磁盘是只读的或者并未挂装，就不存在破坏文件系统或致使文件系统处于不一致状态的风险。另一个可行方法是，通过NFS（network filesystem，网络文件系统）挂装所有的文件系统。这个方法要求内核具有“NFS-Root”的能力，而且在引导时还需传入一些特定的参数。如果采用这种方法，即使我们不借助于SysRq，也能避免任何文件系统的崩溃，因为NFS服务器管理着文件系统的一致性，而它并不受设备驱动程序的影响。

调试器和相关工具

最后一种调试模块的方法就是使用调试器来一步步地跟踪代码，查看变量和计算机寄存器的值。这种方法非常耗时，应该尽量避免。不过，某些情况下通过调试器对代码进行细粒度的分析是很有价值的。

在内核中使用交互式调试器是一个很复杂的问题。出于对系统所有进程的整体利益的考虑，内核在它自己的地址空间中运行。其结果是，许多用户空间下的调试器所提供的常用功能很难用于内核之中，比如断点和单步调试等。本节着眼于调试内核的几种方法；它们每一种都各有利弊。

使用 gdb

gdb 在探究系统内部行为时非常有用。在我们这个层次上，要熟练使用调试器，需要掌握 *gdb* 命令、了解目标平台的汇编代码，还要具备对源代码和优化后的汇编码进行匹配的能力。

启动调试器时必须把内核看作是一个应用程序。除了指定未压缩的内核映像文件名以外，还应该在命令行中提供“core 文件”的名称。对于正在运行的内核，所谓的 core 文件就是这个内核在内存中的核心映像，即 */proc/kcore*。典型的 *gdb* 调用如下所示：

```
gdb /usr/src/linux/vmlinux /proc/kcore
```

第一个参数是未经压缩的内核 ELF 可执行文件的名字，而不是 *zImage* 或 *bzImage* 以及其他任何针对特定引导环境创建的特殊内核映像。

gdb 命令行的第二个参数是 core 文件的名字。与其他 */proc* 中的文件类似，*/proc/kcore* 也是在被读取时产生的。在 */proc* 文件系统中执行 *read* 系统调用时，它会映射到一个用于数据生成而不是数据读取的函数上；我们已在“使用 /proc 文件系统”一节中介绍了这个特性。*kcore* 用来按照 core 文件的格式表示内核的“可执行文件”；由于它要表示对应于所有物理内存的整个内核地址空间，所以是一个非常巨大的文件。在 *gdb* 的使用中，可以通过标准 *gdb* 命令查看内核变量。例如，`p jiffies` 命令可以打印从系统启动到当前时刻的时钟滴答数。

当从 *gdb* 打印数据时，内核仍在运行，不同数据项的值会在不同时刻有所变化；然而，*gdb* 为了优化对 core 文件的访问，会将已经读到的数据缓存起来。如果再次查看 `jiffies` 变量，仍会得到和上次一样的值。对通常的 core 文件来说，对变量值进行缓存是正确的，这样可避免额外的磁盘访问。但对“动态的”core 文件来说就不方便了。解决方法是在需要刷新 *gdb* 缓存的时候，执行 *core-file /proc/kcore* 命令；调试器将使用新的 core 文件

并丢弃所有的旧信息。不过，读取新数据时并不总是需要执行 *core-file* 命令，因为 *gdb* 以几 KB 大小的小数据块形式读取 core 文件，缓存的仅是已经引用的若干小块。

对内核进行调试时，*gdb* 的许多常用功能都不可用。例如，*gdb* 不能修改内核数据；因为在处理其内存映像之前，*gdb* 期望把待调试的程序运行在自己的控制之下。同样，我们也不能设置断点或观察点，或者单步跟踪内核函数。

注意，为了让 *gdb* 使用内核的符号信息，我们必须在打开 CONFIG_DEBUG_INFO 选项的情况下编译内核。其结果将产生一个非常大的内核映像，但若没有符号信息，观察内核变量的目的基本上无法完成。

在调试信息可用的情况下，我们可了解到许多内核内部的工作情况。*gdb* 可以轻松打印结构、跟踪指针等等。但是，困难在于处理模块。因为模块不是传递给 *gdb* 的 *vmlinux* 映像的一部分，因此调试器根本不知道模块的存在。幸运的是，从内核 2.6.7 开始，我们可以通过一些方法告诉 *gdb* 有关可装载模块的信息。

Linux 的可装载模块是 ELF 格式的可执行映像，模块会被划分为许多代码段。一个典型的模块可能包含十多个或者更多的代码段，但对调试会话来讲，相关的代码段只有下面三个：

`.text`

这个代码段包含了模块的可执行代码。调试器必须知道该代码段的位置才能给出追踪信息或者设置断点（当我们在 */proc/kcore* 上运行调试器时，这两个操作均无法实现，但如果使用下面讲到的 *kgdb*，则这两个操作非常有用）。

`.bss`

`.data`

这两个代码段保存模块的变量。任何编译时未初始化的变量保存在 `.bss` 段，而其他经过初始化的变量保存在 `.data` 段。

为了 *gdb* 能够处理可装载模块，必须告诉调试器装载模块代码段的具体位置。该信息可通过 sysfs 的 */sysfs/module* 获得。例如，在装载了 *scull* 模块之后，*/sys/module/scull/sections* 目录中将包含类似 *.text* 这样名字的文件，这些文件的内容是对应代码段的基地址。

现在可以通过一条 *gdb* 命令告诉调试器有关模块的信息了。这条命令就是 `add-symbol-file`，该命令需要用模块目标文件的名称、*.text* 段的基地址以及其他一些选项作为参数，这些选项描述了其他必要的代码段信息。通过 sysfs 获取模块的代码段数据后，我们可以如下构造这条命令：

```
(gdb) add-symbol-file .../scull.ko 0xd0832000 \
        -s .bss 0xd0837100 \
        -s .data 0xd0836be0
```

在示例代码中已经包含了一个小的脚本（*gdbline*），使用这个脚本可以为某个给定的模块构造上述命令。

之后，就可以使用*gdb*来检查可装载模块中的变量了。下面是来自某个*scull*调试会话的示例：

```
(gdb) add-symbol-file scull.ko 0xd0832000 \
      -s .bss 0xd0837100 \
      -s .data 0xd0836be0
add symbol table from file "scull.ko" at
        .text_addr = 0xd0832000
        .bss_addr = 0xd0837100
        .data_addr = 0xd0836be0
(y or n) y
Reading symbols from scull.ko...done.
(gdb) p scull_devices[0]
$1 = {data = 0xcfd66c50,
      quantum = 4000,
      qset = 1000,
      size = 20881,
      access_key = 0,
      ...}
```

从上面的例子看出，第一个*scull*设备目前保存有20 881字节的数据。如果愿意，我们还可以跟踪数据链，或者查看模块中的其他任何感兴趣的数据。另外一个值得掌握的技巧是：

```
(gdb) print *(address)
```

在上面的命令中，我们要为address传入一个十六进制的地址值，该命令的输出是该地址对应的文件以及代码行数。这个技巧非常有用，例如，我们可以利用这条命令找出某个函数指针所指的函数定义在什么地方。

但是，我们仍然不能通过*gdb*执行典型的调试命令，比如设置断点或者修改变量值。为了执行这些操作，需要类似*kdb*（下一节描述）或者*kgdb*（稍后介绍）这样的工具。

kdb内核调试器

很多读者可能会奇怪，为什么不把一些更高级的调试功能直接编译进内核呢。答案很简单，因为Linus不信任交互式的调试器。他担心这些调试器会导致一些不良的修改，也就是说，修补的仅是一些表面现象，而没有发现问题的真正原因所在。因此，他不支持在内核中内置调试器。

然而，其他的内核开发人员偶尔也会用到一些交互式的调试工具。*kdb* 就是其中一种内置的内核调试器，它在 *oss.sgi.com* 上以非正式的补丁形式提供。要使用 *kdb*，必须首先获得这个补丁（取得的版本一定要和内核版本相匹配），然后对当前内核源代码进行 patch 操作，再重新编译并安装这个内核。注意，*kdb* 仅可用于 IA-32（x86）系统（虽然用于 IA-64 的一个版本在主流内核源代码中短暂地出现过，但很快就被删去了）。

一旦运行的是支持 *kdb* 的内核，则可以用下面几个方法进入 *kdb* 的调试状态。在控制台上按下 Pause（或 Break）键将启动调试。当内核发生 oops，或到达某个断点时，也会启动 *kdb*。无论是哪一种情况，都会看到下面这样的消息：

```
Entering kdb (0xc0347b80) on processor 0 due to Keyboard Entry
[0]kdb>
```

注意，当 *kdb* 运行时，内核所做的每一件事情都会停下来。当激活 *kdb* 调试时，系统不应运行其他任何东西；尤其是，不要开启网络功能 —— 当然，除非是在调试网络驱动程序。一般来说，如果要使用 *kdb*，最好在启动时进入单用户模式。

作为一个例子，考虑下面这个快速的 *scull* 调试过程。假定驱动程序已被载入，可以像下面这样指示 *kdb* 在 *scull_read* 函数中设置一个断点：

```
[0]kdb> bp scull_read
Instruction(i) BP #0 at 0xcd087c5dc (scull_read)
    is enabled globally adjust 1
[0]kdb> go
```

bp 命令指示 *kdb* 在内核下一次进入 *scull_read* 时停止运行。随后我们输入 **go** 继续执行。在把一些东西放入 *scull* 的某个设备之后，我们可以在另一台终端的 shell 中运行 *cat* 命令尝试读取这个设备，这样一来就会产生如下的状态：

```
Instruction(i) breakpoint #0 at 0xd087c5dc (adjusted)
0xd087c5dc scull_read:          int3

Entering kdb (current=0xcf09f890, pid 1575) on processor 0 due to
Breakpoint @ 0xd087c5dc
[0]kdb>
```

我们现在正处于 *scull_read* 的开头位置。为了查明是怎样到达这个位置的，我们可以看看堆栈跟踪记录：

```
[0]kdb> bt
    ESP     EIP         Function (args)
0xcdbddf74 0xd087c5dc [scull]scull_read
0xcdbddf78 0xc0150718 vfs_read+0xb8
0xcdbddfa4 0xc01509c2 sys_read+0x42
0xcdbddfc4 0xc0103fcf syscall_call+0x7
[0]kdb>
```

kdb 试图打印出调用跟踪所记录的每个函数的参数列表。然而，它往往会被编译器所使用的优化技巧弄糊涂。因此，它无法正确打印 *scull_read* 的参数。

下面我们来看看如何查询数据。*mds* 命令是用来对数据进行处理的；我们可以用下面的命令查询 scull_devices 指针的值：

```
[0]kdb> mds scull_devices 1
0xd0880de8 cf36ac00    ....
```

在这里，我们要查看的是从 scull_devices 指针位置开始的一个字大小（4 个字节）的数据；该命令的结果告诉我们，设备数组的起始地址位于 0xd0880de8，而第一个设备结构本身位于 0xcf36ac00。要查看设备结构中的数据，我们需要用到这个地址：

```
[0]kdb> mds cf36ac00
0xcf36ac00 ce137dbc ....
0xcf36ac04 00000fa0 ....
0xcf36ac08 000003e8 ....
0xcf36ac0c 0000009b ....
0xcf36ac10 00000000 ....
0xcf36ac14 00000001 ....
0xcf36ac18 00000000 ....
0xcf36ac1c 00000001 ....
```

上面的 8 行数据分别对应于 scull_dev 结构中起始数据。这样，通过这些数据可以知道，第一个设备的内存是从 0xce137dbc 开始分配的，量子大小为 4000（十六进制形式为 fa0）字节，量子集大小为 1000（十六进制形式为 3e8），这个设备中保存有 155（十六进制形式为 9b）个字节的数据，等等。

kdb 还可以修改数据。假设我们要从设备中削减一些数据：

```
[0]kdb> mm cf26ac0c 0x50
0xcf26ac0c = 0x50
```

接下来对设备的 *cat* 操作所返回的数据就会少于上次。

kdb 还有许多其他功能，包括单步调试（根据指令，而不是 C 源代码行），在数据访问中设置断点、反汇编代码、跟踪链表以及访问寄存器数据等等。在应用了 *kdb* 补丁之后，在内核源代码树的 *Documentation/kdb* 目录下可以找到完整的 *kdb* 相关手册页。

kgdb 补丁

目前我们看到了两种交互式调试方法（在 */proc/kcore* 上使用 *gdb* 以及 *kdb*），这两种方法均无法提供类似于应用程序开发人员使用的环境。我们希望有一种真正的内核调试器，它支持变量的修改及断点的设置等等。

如同我们提到过的，的确存在这样一种解决方案。在本书编写时，存在两个独立的补丁，它们均可以让拥有完整功能的 *gdb* 针对内核运行。令人迷惑的是，这两个补丁均称为“kgdb”。它们将运行调试内核的系统和运行调试器的系统隔离开来而工作，而这两个系统之间通过串口线缆连接。因此，开发人员可在他或她的稳定桌面系统上运行 *gdb*，而在“作牺牲用的”测试系统上操作要调试的内核。在这种工作模式下，设置和安装 *gdb* 需要花一些先期功夫，但对调试困难的缺陷来讲，其收益很快就能体现出来。

这两个补丁均处于剧烈变化的状态，而且可能在将来合并到一起，因此我们不打算在这里详细讲述这两个补丁的基本功能。我们鼓励感兴趣的读者跟踪相关事件的发生。第一个 *kgdb* 补丁可在 -mm 内核树（亦即 2.6 主流代码的补丁区）中找到。这个 *kgdb* 版本支持 x86、SuperH、ia64、x96_64、SPAR 以及 32 位的 PPC 架构。除了通过串口操作之外，它还支持通过局域网的通信。通过打开以太网模式，并在引导时设置 `kgdboe` 参数，指出调试命令来源地的 IP 地址就可以支持局域网通信。*Documentation/i386/kgdb* 下的文档描述了如何进行相关设置（注 4）。

另外，读者还可以使用 *http://kgdb.sf.net/* 上的 *kgdb* 补丁。虽然这个 *kgdb* 版本不支持网络通信模式（据说正在开发中），但它对可装载模块提供了内置支持。它支持 x86、x86_64、PowerPC 以及 S/390 架构。

用户模式的 Linux 虚拟机

用户模式 Linux（User-Mode Linux，UML）是一个很有意思的概念。它作为一个独立的、可移植的 Linux 内核而被创建，包含在子目录 *arch/um* 中。然而，它并不是运行在某种新的硬件上，而是运行在基于 Linux 系统调用接口所实现的虚拟机之上。因此，用户模式 Linux 可以使 Linux 内核成为一个运行在 Linux 系统之上的、独立的用户模式的进程。

将一个内核副本当作用户模式下的进程来运行可以带来很多好处。因为它运行在一个受约束的虚拟处理器之上，所以有错误的内核不会破坏“真正的”系统。对软 / 硬件的不同配置可以在相同的框架中轻易地进行尝试。并且，对于内核开发人员来说最值得关注的特点在于，可以很容易地利用 *gdb* 或其他调试器对用户模式 Linux 进行处理，因为归根结底它只是一个进程。很明显，UML 有潜力加快内核的开发过程。

注 4：然而忽略指出的是，我们应该将网络适配器的驱动程序构建到内核中，否则调试器无法在引导阶段找到网络适配器，从而会自动关机。

然而，从驱动程序编写者的角度看，UML也有非常明显的缺点：用户模式的内核无法访问主机系统的硬件。这样，虽然通过UML可以调试本书中提到的大部分示例驱动程序但却无法调试那些和真正的硬件打交道的驱动程序。

有关UML的详细信息，可访问*http://user-mode-linux.sf.net/*。

Linux跟踪工具包

Linux跟踪工具包（Linux Trace Toolkit，LTT）是一个内核补丁，包含了一组可以用于内核事件跟踪的相关工具集。跟踪内容包括时间信息，而且还能合理地建立在一段指定时间内所发生事件的完整描述。因此，LTT不仅能用于调试，还能用来捕捉性能方面的问题。

在*http://www.opersys.com/LTT*上可以找到LTT以及大量的文档资料。

动态探测

动态探测（Dynamic Probes，DProbes）是IBM为基于IA-32架构的Linux发布的一种调试工具（遵循GPL协议）。它可在系统的几乎任何一个地方放置一个“探针”，既可以是用户空间也可以是内核空间。这个探针由一些特殊代码（用一种特别设计的、面向堆栈的语言编写）组成，当控制到达给定点时，这些代码开始执行。这种代码能向用户空间汇报数据、修改寄存器，或者完成许多其他工作。DProbes很有用的特点是，一旦内核中编译进了这个功能，探针就可以插到一个运行系统的任意一个位置，而无需重建内核或重新启动。DProbes也可以协同LTT工具在任意位置插入新的跟踪事件。

DProbes工具可以从IBM的开放源代码站点*http://oss.software.ibm.com*上下载。



第五章

并发和竞态

目前为止，我们很少关注并发问题——亦即，当系统试图一次完成多个任务时会产生什么结果。但是，对并发的管理是操作系统编程中核心的问题之一。并发相关的缺陷是最容易制造的，也是最难找到的。即使是Linux内核开发专家也会偶尔制造并发相关的缺陷。

在早期的Linux内核中，并发的来源相对较少。早期内核不支持对称多处理（symmetric multiprocessing，SMP），因此，导致并发执行的唯一原因是对硬件中断的服务。这种情况处理起来较为简单，但并不适用于为获得更好的性能而使用更多处理器且强调快速响应事件的系统。为了响应现代硬件和应用程序的需求，Linux内核已经发展到同时处理更多事情的时代。这种变革使得内核性能及伸缩性得到了相当大的提高，然而也极大提高了内核编程的复杂性。设备驱动程序开发者必须在开始设计时就考虑到并发因素，并且还必须对内核提供的并发管理设施有坚实的理解。

本章将帮助读者对并发管理有个初步的理解。为此，我们将介绍一些用于并发管理的设施，同时将应用来自第三章的*scull*驱动程序，而本章介绍的其他设施不会在这里使用。但是首先，我们要分析简单的*scull*驱动程序可能会导致什么问题，并介绍如何避免这些潜在的问题。

scull的缺陷

首先快速浏览*scull*内存管理代码的一些片断。深入到驱动程序的*write*逻辑时，我们发现，*scull*必须判断所请求的内存是否已经分配好。下面的代码处理了这个问题：

```
if (!dptr->data[s_pos]) {
    dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
    if (!dptr->data[s_pos])
        goto out;
}
```

假定有两个进程（我们称之为“A”和“B”）正在独立地尝试向同一个*scull*设备的相同偏移量写入数据，而且两个进程在同一时刻到达上述代码段中的第一个`if`判断语句。如果代码涉及的指针是NULL，两个进程都会决定分配内存，而每个进程都会将结果指针赋值给`dptr->data[s_pos]`。因为两个进程对同一位置赋值，显然只有一个赋值会成功。

当然，其结果是第二个完成赋值的进程会“胜出”。如果进程A首先赋值，则它的赋值会被进程B覆盖。这样，*scull*会完全忘记由A分配的内存，而只会记录由进程B分配得到的指针。因此，由A分配的内存将丢失，从而永远不会返回到系统中。

上述事件过程就是一种竞态（race condition）。竞态会导致对共享数据的非控制访问。发生错误的访问模式时，会产生非预期的结果。对这里讨论的竞态，其结果是内存的泄漏。这种结构已经够糟糕的了，但某些竞态经常会导致系统崩溃、数据被破坏或者产生安全问题。因为竞态是一种极端低可能性的事件，因此程序员往往会忽视竞态。但是在计算机世界中，百万分之一的事件可能没几秒就会发生，而其结果是灾难性的。

稍后我们会消除来自*scull*的竞态，但是首先需要对并发有个更一般性的描述。

并发及其管理

在现代Linux系统中存在大量的并发来源，因此会导致可能的竞态。正在运行的多个用户空间进程可能以一种令人惊讶的组合方式访问我们的代码。SMP系统甚至可在不同的处理器上同时执行我们的代码。内核代码是可抢占的；因此，我们的驱动程序代码可能在任何时候丢失对处理器的独占，而拥有处理器的进程可能正在调用我们的驱动程序代码。设备中断是异步事件，也会导致代码的并发执行。内核还提供了许多可延迟代码执行的机制，比如workqueue（工作队列）、tasklet（小任务）以及timer（定时器）等，这些机制使得代码可在任何时刻执行，而不管当前进程在做什么。在现代的热插拔世界中，设备可能会在我们正使用时消失。

对竞态的避免也许是一种胁迫性的任务。在任何事情可在任何时间发生的世界中，驱动程序开发者如何才能避免制造这种混乱状态？如我们所愿，大部分竞态可通过使用内核的并发控制原语，并应用几个基本的原理来避免。我们首先介绍这些原理，然后讲述如何应用这些原理的细节。

竞态通常作为对资源的共享访问结果而产生。当两个执行线程（注1）需要访问相同的

注1：对本章而言，执行的“线程”指正在运行代码的任意上下文。每个进程显然就是一个执行的线程，但是中断处理例程以及其他用于响应异步内核事件的其他代码也一样是线程。

数据结构（或硬件资源）时，混合的可能性就永远存在。因此在设计自己的驱动程序时，第一个要记住的规则是，只要可能，就应该避免资源的共享。如果没有并发的访问，也就不会有竞态的产生。因此，仔细编写的内核代码应该具有最少的共享。这种思想的最明显应用就是避免使用全局变量。如果我们将资源放在多个执行线程都会找到的地方，则必须有足够的理由。

但是事情的本质是，这种类型的共享通常是必需的。硬件资源本质上就是共享的，而软件资源经常需要对其他执行线程可用。我们还要清楚的是，全局变量并不是共享数据的唯一途径，只要我们的代码将一个指针传递给了内核的其他部分，一个新的共享就可能建立。因此，共享就是现实的生活。

下面是资源共享的硬规则：在单个执行线程之外共享硬件或软件资源的任何时候，因为另外一个线程可能产生对该资源的不一致观察，因此必须显式地管理对该资源的访问。在上面的*scull*示例中，从进程B的角度所看到的数据是不一致的，也就是说，进程B不知道进程A已经为该（共享）设备分配了内存，因此它会执行它自己的分配并覆盖A的工作。在这种情况下，我们必须控制对*scull*数据结构的访问。我们需要重新设计，使得代码要么看到已经分配好的内存，要么知道内存还没有分配或将要由其他人分配。访问管理的常见技术称为“锁定”或者“互斥”——确保一次只有一个执行线程可操作共享资源。本章其余的大部分内容将讲述锁定机制。

但是，我们首先必须简要考虑另外一个重要的规则。当内核代码创建了一个可能和其他内核部分共享的对象时，该对象必须在还有其他组件引用自己时保持存在（并正确工作）。当*scull*让自己的设备可用之时，它必须准备好在其设备上的请求，直到这些设备上不存在任何引用（比如已打开的用户空间文件）为止。这一规则产生下面两个需求：在对象尚不能正确工作时，不能将其对内核可用，也就是说，对这类对象的应用必须得到跟踪。在大多数情况下，我们将发现内核会为我们处理引用计数，然而总是会有例外。

遵守上述规则需要仔细关注和规划对细节的处理。如果我们自己还没有认识到对被共享资源的并发访问，则其结果很容易让人迷惑不解。但是通过一些手段，大部分竞态可在其对我们或者我们的用户造成伤害前被处理掉。

信号量和互斥体

接下来我们研究如何为*scull*添加锁定。我们的目的是使对*scull*数据结构的操作是原子的，这意味着在涉及到其他执行线程之前，整个操作就已经结束了。对我们的内存泄漏示例来说，需要确保当一个线程发现特定内存块需要分配时，它应该拥有执行分配的机

会，并需要在其他线程执行同一测试之前完成这个工作。为此，我们必须建立临界区：在任意给定的时刻，代码只能被一个线程执行。

并不是所有的临界区都是一样的，因此内核为不同的需求提供了不同的原语。在我们的例子中，每个发生在进程上下文的对*scull*数据结构的访问都被认为是一个直接的用户请求，来自中断处理例程或者其他异步上下文的访问都不能发生。这里没有特殊的延迟（响应时间）需求，应用程序开发者能够理解I/O请求通常不会立刻得到满足。此外，在访问自己的数据结构时，*scull*并不拥有任何其他关键的系统资源。这一切意味着，当*scull*驱动程序在等待访问数据结构而进入休眠时，不需要考虑其他内核组件。

在这个上下文中，“进入休眠”是一个具有明确定义的术语。当一个Linux进程到达某个时间点，此时它不能进行任何处理时，它将进入休眠（或“阻塞”）状态，这将把处理器让给其他执行线程直到将来它能够继续完成自己的处理为止。在等待I/O完成时，进程经常会进入休眠状态。随着我们对内核理解的深入，将遇到大量不能休眠的情况。但是，*scull*中的*write*方法并不是这种情况之一。因此，我们可以使用一种锁定机制，当进程在等待对临界区的访问时，此机制可让进程进入休眠状态。

重要的是，我们将执行另一个操作（使用*kmalloc*分配内存），该操作也可能会休眠，因此休眠可能在任何时刻发生。为了让我们的临界区正确工作，我们选择使用的锁定原语必须在其他拥有这个锁并休眠的情况下工作。在可能出现休眠的情况下，并不是所有的锁定机制都可用（稍后我们将看到一些不能休眠的锁机制）。而目前，对于我们来说最合适的机制是信号量（semaphore）。

在计算机科学中，信号量是一个众所周知的概念。一个信号量本质上是一个整数值，它和一对函数联合使用，这一对函数通常称为*P*和*V*。希望进入临界区的进程将在相关信号量上调用*P*；如果信号量的值大于零，则该值会减小一，而进程可以继续。相反，如果信号量的值为零（或更小），进程必须等待直到其他人释放该信号量。对信号量的解锁通过调用*V*完成；该函数增加信号量的值，并在必要时唤醒等待的进程。

当信号量用于互斥时（即避免多个进程同时在一个临界区中运行），信号量的值应初始化为1。这种信号量在任何给定时刻只能由单个进程或线程拥有。在这种使用模式下，一个信号量有时也称为一个“互斥体（mutex）”，它是互斥（mutual exclusion）的简称。Linux内核中几乎所有的信号量均用于互斥。

Linux信号量的实现

Linux内核遵守上述语义提供了信号量的实现，然而在术语上存在一些差异。要使用信号量，内核代码必须包括<*asm/semaphore.h*>。相关的类型是`struct semaphore`；实

际的信号量可通过几种途径来声明和初始化。其中之一是直接创建信号量，这通过*sema_init*完成：

```
void sema_init(struct semaphore *sem, int val);
```

其中val是赋予一个信号量的初始值。

不过，信号量通常被用于互斥模式。为了让这种常见情况更加简单，内核提供了一组辅助函数和宏。因此，我们可以用下面的方法之一来声明和初始化一个互斥体：

```
DECLARE_MUTEX(name);
DECLARE_MUTEX_LOCKED(name);
```

上面两个宏的结果是，一个称为name的信号量变量被初始化为1（使用DECLARE_MUTEX）或者0（使用DECLARE_MUTEX_LOCKED）。在后面一种情况下，互斥体的初始状态是锁定的，也就是说，在允许任何线程访问之前，必须显式地解锁该互斥体。

如果互斥体必须在运行时被初始化（例如在动态分配互斥体的情况下），应使用下面的函数之一：

```
void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(struct semaphore *sem);
```

在Linux世界中，*P*函数被称为*down*——或者这个名字的其他变种。这里，“down”指的是该函数减小了信号量的值，它也许会将调用者置于休眠状态，然后等待信号量变得可用，之后授予调用者对被保护资源的访问。下面是*down*的三个版本：

```
void down(struct semaphore *sem);
int down_interruptible(struct semaphore *sem);
int down_trylock(struct semaphore *sem);
```

*down*减小信号量的值，并在必要时一直等待。*down_interruptible*完成相同的工作，但操作是可中断的。可中断的版本几乎是我们始终要使用的版本，它允许等待在某个信号量上的用户空间进程可被用户中断。作为通常的规则，我们不应该使用非中断操作，除非没有其他可变通的办法。非中断操作是建立不可杀进程（*ps*输出中的“D state”）的好方法，但会让用户感到懊恼。使用*down_interruptible*需要额外小心，如果操作被中断，该函数会返回非零值，而调用者不会拥有该信号量。对*down_interruptible*的正确使用需要始终检查返回值，并作出相应的响应。

最后一个版本（*down_trylock*）永远不会休眠；如果信号量在调用时不可获得，*down_trylock*会立即返回一个非零值。

当一个线程成功调用上述*down*的某个版本之后，就称为该线程“拥有”（或“拿到”、“获

取”）了该信号量。这样，该线程就被赋予访问由该信号量保护的临界区的权利。当互斥操作完成后，必须返回该信号量。Linux 等价于 *V* 的函数是 *up*：

```
void up(struct semaphore *sem);
```

调用 *up* 之后，调用者不再拥有该信号量。

如读者所料，任何拿到信号量的线程都必须通过一次（只有一次）对 *up* 的调用而释放该信号量。在出现错误的情况下，经常需要特别小心；如果在拥有一个信号量时发生错误，必须在将错误状态返回给调用者之前释放该信号量。我们很容易犯忘记释放信号量的错误，而其结果（进程在某些无关位置处被挂起）很难复现和跟踪。

在 scull 中使用信号量

信号量机制为 *scull* 提供了一种工具，它可以利用信号量避免在访问 `scull_dev` 数据结构时产生竞态。但我们必须正确使用这个工具。正确使用锁定机制的关键是，明确指定需要保护的资源，并确保每一个对这些资源的访问使用正确的锁定。在我们的示例驱动程序中，所有的信息都包含在 `scull_dev` 结构中，因此该结构就是我们锁定机构的逻辑范围。

该结构的定义如下：

```
struct scull_dev {
    struct scull_qset *data;  /* 指向第一个量子集的指针 */
    int quantum;              /* 当前的量子大小 */
    int qset;                 /* 当前的数组大小 */
    unsigned long size;       /* 保存在其中的数据总量 */
    unsigned int access_key;  /* 由 sculluid 和 scullpriv 使用 */
    struct semaphore sem;     /* 互斥信号量 */
    struct cdev cdev;         /* 字符设备结构       */
};
```

该结构底部有一个称为 `sem` 的成员，它就是我们的信号量。我们决定对每个虚拟的 *scull* 设备使用单独的信号量。使用单个全局的信号量也是正确的。但是不同的 *scull* 设备并不共享资源，因此没有理由让一个进程在其他进程访问不同的 *scull* 设备时等待。为每个设备使用单独的信号量允许不同设备上的操作可以并行处理，从而可以提高性能。

信号量在使用前必须初始化。*scull* 在装载时通过下面的循环执行初始化：

```
for (i = 0; i < scull_nr_devs; i++) {
    scull_devices[i].quantum = scull_quantum;
    scull_devices[i].qset = scull_qset;
    init_MUTEX(&scull_devices[i].sem);
    scull_setup_cdev(&scull_devices[i], i);
}
```

注意，信号量必须在*scull*设备对系统其他部分可用前被初始化。因此，我们在*scull_setup_cdev*之前调用了*init_MUTEX*。以相反的顺序执行上述操作会建立一个竞态，即在信号量准备好之前，有代码可能访问它们。

接下来，我们必须仔细检查代码，确保在不拥有该信号量的时候不会访问scull_dev数据结构。例如，*scull_write*的开始处包含下面的代码：

```
if (down_interruptible(&dev->sem))
    return -ERESTARTSYS;
```

注意代码中对*down_interruptible*返回值的检查；如果它返回非零值，则说明操作被中断。这种情况下，通常要做的工作是返回-ERESTARTSYS。在见到这个返回代码后，内核的高层代码要么会从头重新启动该调用，要么会将该错误返回给用户。如果我们返回-ERESTARTSYS，则必须首先撤销已经做出的任何用户可见的修改，这样，系统调用可正确重试。如果无法撤销这些操作，则应该返回-EINTR。

不管*scull_write*是否能够成功完成其他工作，它都必须释放信号量。如果一切正常，执行过程将到达该函数的最后几行：

```
out:
  up(&dev->sem);
  return retval;
```

上述代码释放信号量，并返回被调用的状态值。*scull_write*中有几个地方可能会产生错误，这包括内存分配失败，或者在试图从用户空间复制数据时产生故障等。在这些情况下，代码会执行goto out，这样可以确保正确完成清除工作。

读取者/写入者信号量

信号量对所有的调用者执行互斥，而不管每个线程到底想做什么。但是，许多任务可以划分为两种不同的工作类型：一些任务只需要读取受保护的数据结构，而其他的则必须做出修改。允许多个并发的读取者是可能的，只要它们之中没有哪个要做修改。这样做可以大大提高性能，因为只读任务可并行完成它们的工作，而不需要等待其他读取者退出临界区。

Linux内核为这种情形提供了一种特殊的信号量类型，称为“rwsem”（或者“reader/writer semaphore，读取者/写入者信号量”）。在驱动程序中使用rwsem的机会相对较少，但偶尔也比较有用。

使用rwsem的代码必须包括*<linux/rwsem.h>*。读取者/写入者信号量相关的数据类型是struct rw_semaphore；一个rwsem对象必须在运行时通过下面的函数显式地初始化：

```
void init_rwsem(struct rw_semaphore *sem);
```

新初始化的rwsem可用于其后出现的任务（读取者或写入者）。对只读访问，可用的接口如下：

```
void down_read(struct rw_semaphore *sem);
int down_read_trylock(struct rw_semaphore *sem);
void up_read(struct rw_semaphore *sem);
```

对*down_read*的调用提供了对受保护资源的只读访问，可和其他读取者并发地访问。注意，*down_read*可能会将调用进程置于不可中断的休眠。*down_read_trylock*不会在读取访问不可获得时等待；它在授予访问时返回非零，其他情况下返回零。注意，*down_read_trylock*的用法和其他大多数内核函数不同，其他函数会在成功时返回零。由*down_read*获得的rwsem对象最终必须通过*up_read*被释放。

针对写入者的接口类似于读取者接口：

```
void down_write(struct rw_semaphore *sem);
int down_write_trylock(struct rw_semaphore *sem);
void up_write(struct rw_semaphore *sem);
void downgrade_write(struct rw_semaphore *sem);
```

down_write、*down_write_trylock*和*up_write*与读取者的对应函数行为相同，当然，它们提供的是写入访问。当某个快速改变获得了写入者锁，而其后是更长时间的只读访问的话，我们可以在结束修改之后调用*downgrade_write*，来允许其他读取者的访问。

一个rwsem可允许一个写入者或无限多个读取者拥有该信号量。写入者具有更高的优先级；当某个给定写入者试图进入临界区时，在所有写入者完成其工作之前，不会允许读取者获得访问。如果有大量的写入者竞争该信号量，则这种实现会导致读取者“饿死”，即可能会长期拒绝读取者的访问。为此，最好在很少需要写访问且写入者只会短期拥有信号量的时候使用rwsem。

completion

内核编程中常见的一种模式是，在当前线程之外初始化某个活动，然后等待该活动的结束。这个活动可能是，创建一个新的内核线程或者新的用户空间进程、对一个已有进程的某个请求，或者某种类型的硬件动作，等等。在这种情况下，我们可以使用信号量来同步这两个任务，并如下所示来编写代码：

```
struct semaphore sem;

init_MUTEX_LOCKED(&sem);
start_external_task(&sem);
down(&sem);
```

当外部任务完成其工作时，将调用up(&sem)。

但信号量并不是适用这种情况的最好工具。在通常的使用中，试图锁定某个信号量的代码会发现该信号量几乎总是可用；而如果存在针对该信号量的严重竞争，性能将受到影响，这时，我们需要重新审视锁定机制。因此，信号量对“可用”情况已经做了大量优化。然而，如果像上面那样使用信号量在任务完成时进行通信，则调用*down*的线程几乎总是要等待，这样性能也同样会受到影响。如果信号量在这种情况下声明为自动变量，则也可能受某个（难对付的）竞态的影响。在某些情况下，信号量可能在调用*up*的进程完成其相关任务前消失。

上述考虑导致2.4.7版内核中出现了“completion（完成）”接口。completion是一种轻量级的机制，它允许一个线程告诉另一线程某个工作已经完成。为了使用completion，代码必须包含*<linux/completion.h>*。可以利用下面的接口创建completion：

```
DECLARE_COMPLETION(my_completion);
```

或者，如果必须动态地创建和初始化completion，则使用下面的方法：

```
struct completion my_completion;
/* ... */
init_completion(&my_completion);
```

要等待completion，可进行如下调用：

```
void wait_for_completion(struct completion *c);
```

注意，该函数执行一个非中断的等待。如果代码调用了*wait_for_completion*且没有人会完成该任务，则将产生一个不可杀的进程（注2）。

另一方面，实际的completion事件可通过调用下面函数之一来触发：

```
void complete(struct completion *c);
void complete_all(struct completion *c);
```

这两个函数在是否有多个线程在等待相同的completion事件上有所不同。*complete*只会唤醒一个等待线程，而*complete_all*允许唤醒所有等待线程。在大多数情况下，只会有一个等待者，因此这两个函数产生相同的结果。

一个completion通常是一个单次（one-shot）设备；也就是说，它只会被使用一次然后被丢弃。但是，如果仔细处理，completion结构也可以被重复使用。如果没有使用*complete_all*，则我们可以重复使用一个completion结构，只要那个将要触发的事件是

注2：在本书编写时，添加可中断版本的补丁已进入测试周期，但尚未合并到主线内核。

明确而不含糊的，就不会带来任何问题。但是，如果使用了*complete_all*，则必须在重复使用该结构之前重新初始化它。下面这个宏可用来快速执行重新初始化：

```
INIT_COMPLETION(struct completion c);
```

作为completion的使用方法的示例，可参阅示例源代码中的*complete*模块。该模块定义了一个语义非常简单的设备：任何试图从该设备读取的进程都将等待（使用*wait_for_completion*），直到其他进程写入该设备为止。实现这种行为的代码如下：

```
DECLARE_COMPLETION(comp);

ssize_t complete_read (struct file *filp, char _ _user *buf, size_t count,
loff_t *pos)
{
    printk(KERN_DEBUG "process %i (%s) going to sleep\n",
           current->pid, current->comm);
    wait_for_completion(&comp);
    printk(KERN_DEBUG "awoken %i (%s)\n", current->pid, current->comm);
    return 0; /* EOF */
}

ssize_t complete_write (struct file *filp, const char _ _user *buf, size_t
count,
        loff_t *pos)
{
    printk(KERN_DEBUG "process %i (%s) awakening the readers...\n",
           current->pid, current->comm);
    complete(&comp);
    return count; /* 成功，以免重复 */
}
```

同一时刻有多个进程从该设备“读取”是可能的。每次向该设备的写入将导致一个读取操作结束，但是没有办法知道会是哪个进程。

completion机制的典型使用是模块退出时的内核线程终止。在这种原型中，某些驱动程序的内部工作由一个内核线程在while (1)循环中完成。当内核准备清除该模块时，exit函数会告诉该线程退出并等待completion。为了实现这个目的，内核包含了可用于这种线程的一个特殊函数：

```
void complete_and_exit(struct completion *c, long retval);
```

自旋锁

信号量对互斥来讲是非常有用的工具，但它并不是内核提供的唯一的这类工具。相反，大多数锁定通过称为“自旋锁（spinlock）”的机制实现。和信号量不同，自旋锁可在不

能休眠的代码中使用，比如中断处理例程。在正确使用的情况下，自旋锁通常可以提供比信号量更高的性能。但是，自旋锁也带来了其他一组不同的使用限制。

在概念上，自旋锁非常简单。一个自旋锁是一个互斥设备，它只能有两个值："锁定"和"解锁"。它通常实现为某个整数值中的单个位。希望获得某特定锁的代码测试相关的位。如果锁可用，则"锁定"位被设置，而代码继续进入临界区；相反，如果锁被其他人获得，则代码进入忙循环并重复检查这个锁，直到该锁可用为止。这个循环就是自旋锁的"自旋"部分。

当然，自旋锁的真实实现要比上面描述的复杂一些。"测试并设置"的操作必须以原子方式完成，这样，即使有多个线程在给定时间自旋，也只有一个线程可获得该锁。在超线程处理器上，还必须仔细处理以避免死锁。这里的超线程处理器可实现多个虚拟的CPU，它们共享单个处理器核心及缓存。因此，实际的自旋锁实现由于Linux所支持的架构的不同而不同。但是，核心概念对所有系统来讲是一样的，当存在自旋锁时，等待执行忙循环的处理器做不了任何有用的工作。

自旋锁最初是为了在多处理器系统上使用而设计的。只要考虑到并发问题，单处理器工作站在运行可抢占内核时其行为就类似于SMP。如果非抢占式的单处理器系统进入某个锁上的自旋状态，则会永远自旋下去；也就是说，没有任何其他线程能够获得CPU来释放这个锁。出于对此原因的考虑，非抢占式的单处理器系统上的自旋锁被优化为不做任何事情，但改变IRQ掩码状态的例程是个例外。因为抢占，即使不打算在SMP系统上运行自己的代码，我们仍然需要实现正确的锁定。

自旋锁API介绍

自旋锁原语所需要包含的文件是*<linux/spinlock.h>*。实际的锁具有`spinlock_t`类型。和其他任何数据结构类似，一个自旋锁必须被初始化。对自旋锁的初始化可在编译时通过下面的代码完成：

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
```

或者在运行时，调用下面的函数：

```
void spin_lock_init(spinlock_t *lock);
```

在进入临界区之前，我们的代码必须调用下面的函数获得需要的锁：

```
void spin_lock(spinlock_t *lock);
```

注意，所有的自旋锁等待在本质上都是不可中断的。一旦调用了*spin_lock*，在获得锁之前将一直处于自旋状态。

要释放已经获取的锁，可将锁传递给下面的函数：

```
void spin_unlock(spinlock_t *lock);
```

还有其他许多自旋锁函数，我们很快就会看到这些函数。但是所有这些函数都离不开上述函数表达的核心思想。除了锁定和释放之外，我们对一个自旋锁本身能做的事情太少。然而，在使用自旋锁上有一些必须遵守的规则。在讲述完整的自旋锁接口之前，我们首先花点时间来了解一下这些规则。

自旋锁和原子上下文

假定我们的驱动程序获得了一个自旋锁，然后在临界区开始了它的工作。在这个过程中间，驱动程序丢掉了处理器。也许它调用了一个函数（比如*copy_from_user*），这个函数使进程进入休眠状态。或者，也许发生了内核抢占，更高优先级的进程将我们的代码排挤到了一边。这样，我们的代码将拥有这个自旋锁，并且在可预见的未来，它不会释放任何时间。如果其他某个线程试图获得相同的锁，在最好的情况下，该线程要等待（在处理器上自旋）很长时间。在最坏的情况下，系统将整个进入死锁状态。

大多数读者都会同意应该避免这种现象。因此，适用于自旋锁的核心规则是：任何拥有自旋锁的代码都必须是原子的。它不能休眠，事实上，它不能因为任何原因放弃处理器，除了服务中断以外（某些情况下此时也不能放弃处理器）。

内核抢占的情况由自旋锁代码本身处理。任何时候，只要内核代码拥有自旋锁，在相关处理器上的抢占就会被禁止。甚至在单处理器系统上，也必须以同样的方式禁止抢占以避免竞态。这就是为什么即使我们不打算在多处理器系统上运行自己的代码，却仍然要正确处理锁定的原因。

在拥有锁的时候避免休眠有时很难做到；许多内核函数可以休眠，而且此行为也始终没有文档来很好地说明。在用户空间和内核空间之间复制数据就是个明显的例子：在复制继续前，必需的用户空间页也许需要从磁盘上交换进入，而这个操作明显需要休眠。需要分配内存的任何操作也会休眠，比如*kmalloc*，如果没有明确告知，它会在等待可用内存时放弃处理器进入休眠。休眠可发生在许多无法预期的地方；当我们编写需要在自旋锁下执行的代码时，必须注意每一个所调用的函数。

还有另外一种情形：我们的驱动程序正在执行，并且已经获得了一个锁，这个锁控制着对设备的访问。在拥有这个锁的时候，设备产生了一个中断，它导致中断处理例程被调用。而中断处理例程在访问设备之前，也要获得这个锁。在中断处理例程中拥有锁是合法的，这也是为什么自旋锁操作不能休眠的一个原因。但是，当中断例程在最初拥有锁

的代码所在的处理器上运行时，会发生什么情况呢？在中断例程自旋时，非中断代码将没有任何机会来释放这个锁。处理器将永远自旋下去。

为了避免这种陷阱，我们需要在拥有自旋锁时禁止中断（仅在本地CPU上）。用于禁止中断的自旋锁函数有许多变种（我们将在下一小节看到这些函数）。但是，对中断的完整讨论将在第十章中展开。

自旋锁使用上的最后一个重要规则是，自旋锁必须在可能的最短时间内拥有。拥有自旋锁的时间越长，其他处理器不得不自旋以等待释放该自旋锁的时间就越长，而它不得不永远自旋的可能性就越大。长的锁拥有时间将阻止对当前处理器的调度，这意味着更高优先级的进程（真正应该获得CPU的进程）不得不等待。为了降低内核的延迟（进程等待调度的时间），内核开发者在2.5开发系列版本中已经花费了大量精力。而一个编写得不好的驱动程序将因为过长时间拥有自旋锁而抹煞这种努力。为了避免制造这种类型的问题，请谨记拥有锁的时间越短越好。

自旋锁函数

我们已经看到两个操作自旋锁的函数：*spin_lock*和*spin_unlock*。但是，还有其他一些具有类似名称和用途的函数。这里我们将给出完整的自旋锁函数。对自旋锁API的彻底理解需要首先理解中断处理以及相关概念，因此，这里的讨论也许会让读者感到迷惑。

锁定一个自旋锁的函数实际有四个：

```
void spin_lock(spinlock_t *lock);
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
void spin_lock_irq(spinlock_t *lock);
void spin_lock_bh(spinlock_t *lock)
```

我们已经知道了*spin_lock*的功能。*spin_lock_irqsave*会在获得自旋锁之前禁止中断（只在本地处理器上），而先前的中断状态保存在`flags`中。如果我们能够确保没有任何其他代码禁止本地处理器的中断（或者换句话说，我们能够确保在释放自旋锁时应该启用中断），则可以使用*spin_lock_irq*，而无需跟踪标志。最后，*spin_lock_bh*在获得锁之前禁止软件中断，但是会让硬件中断保持打开。

如果我们有一个自旋锁，它可以被运行在（硬件或软件）中断上下文中的代码获得，则必须使用某个禁止中断的*spin_lock*形式，因为使用其他的锁定函数迟早会导致系统死锁。如果我们不会在硬件中断处理例程中访问自旋锁，但可能在软件中断（例如，以tasklet的形式运行的代码，第七章讨论该主题）中访问，则应该使用*spin_lock_bh*，以便在安全地避免死锁的同时还能服务硬件中断。

释放自旋锁的方法也有四种，严格对应于获取自旋锁的那些函数：

```
void spin_unlock(spinlock_t *lock);
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
void spin_unlock_irq(spinlock_t *lock);
void spin_unlock_bh(spinlock_t *lock);
```

每个*spin_unlock*的变种都会撤销对应的*spin_lock*函数所做的工作。传递到*spin_unlock_irqrestore*的flags参数必须是传递给*spin_lock_irqsave*的同一个变量。我们还必须在同一个函数中调用*spin_lock_irqsave*和*spin_unlock_irqrestore*，否则代码可能在某些架构上出现问题。

还有如下非阻塞的自旋锁操作：

```
int spin_trylock(spinlock_t *lock);
int spin_trylock_bh(spinlock_t *lock);
```

这两个函数在成功（即获得自旋锁）时返回非零值，否则返回零。对禁止中断的情况，没有对应的“try”版本。

读取者/写入者自旋锁

内核提供自旋锁的读取者/写入者形式，这种自旋锁和本章早先介绍过的读取者/写入者信号量非常相似。这种锁允许任意数量的读取者同时进入临界区，但写入者必须互斥访问。读取者/写入者锁具有rwlock_t类型，在*<linux/spinlock.h>*中定义。我们可以用下面的两种方式声明和初始化它们：

```
rwlock_t my_rwlock = RW_LOCK_UNLOCKED; /* Static way */

rwlock_t my_rwlock;
rwlock_init(&my_rwlock);  /* Dynamic way */
```

可用函数的清单现在看起来应该非常熟悉。对读取者来讲，可使用如下函数：

```
void read_lock(rwlock_t *lock);
void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
void read_lock_irq(rwlock_t *lock);
void read_lock_bh(rwlock_t *lock);

void read_unlock(rwlock_t *lock);
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void read_unlock_irq(rwlock_t *lock);
void read_unlock_bh(rwlock_t *lock);
```

有意思的是，这里并没有*read_trylock*函数可用。

用于写入者的函数类似于读取者，如下所示：

```
void write_lock(rwlock_t *lock);
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
void write_lock_irq(rwlock_t *lock);
void write_lock_bh(rwlock_t *lock);
int write_trylock(rwlock_t *lock);

void write_unlock(rwlock_t *lock);
void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void write_unlock_irq(rwlock_t *lock);
void write_unlock_bh(rwlock_t *lock);
```

和rwsem类似，读取者/写入者锁可能造成读取者饥饿。这种情况几乎不成问题，但是如果对锁的竞争导致饥饿，性能会变得很低。

锁陷阱

多年使用锁的经验（以及那些早于Linux的出现就已获得的经验）说明，我们很难驾轻就熟地使用锁。并发的管理本来就非常棘手，而许多使用方法都可能导致错误。在这一小节中，我们将快速浏览可能导致错误的东西。

不明确的规则

如上所述，恰当的锁定模式需要清晰和明确的规则。当我们创建了一个可被并行访问的对象时，应该同时定义用来控制访问的锁。锁定模式必须在一开始就安排好，否则其后的改进将会非常困难。先期的时间投入通常会在调试阶段获得收益。

在编写代码时肯定会遇到几个函数，它们均需要访问某个受特定锁保护的结构。这时，我们必须小心：如果某个获得锁的函数要调用其他同样试图获取这个锁的函数，我们的代码就会死锁。不论是信号量还是自旋锁，都不允许锁拥有者第二次获得这个锁；如果试图这么做，系统将挂起。

为了让锁定正确工作，则不得不编写一些函数，这些函数假定调用者已经获取了相关的锁。通常，内部的静态函数可通过这种方式编写，而提供给外部调用的函数则必须显式地处理锁定。在编写那些假定调用者已处理了锁定的内部函数时，我们自己应该显式地说明这种假定。因为如果在几个月之后再回头来看这些代码时，我们会发现很难记清在调用某个特定函数时是否需要拥有锁。

在*scull*的例子中，我们所作的设计决策是：由系统调用直接调用的那些函数均要获得信号量，以便保护要访问的设备结构。而其他的内部函数只会由其他的*scull*函数调用，则假定信号量已经被正确获得。

锁的顺序规则

使用大量锁的系统（内核就是这样一个系统）中，代码通常需要一次拥有多个锁。如果某种类型的计算必须使用两个不同的资源来完成，而每个资源都有自己的锁，则通常没有其他方法来同时获取这两个锁。

但是，拥有多个锁可能很危险。如果我们有两个锁，分别是*Lock1*和*Lock2*，而代码需要同时拥有这两个锁，这时就有可能进入潜在的死锁。想像某个线程锁定了*Lock1*，而其他线程同时锁定了*Lock2*。这时，每个线程都试图获得另外的那个锁，于是两个线程都将死锁。

对于这个问题的解决办法通常比较简单：在必须获取多个锁时，应该始终以相同的顺序获得。只要遵守这个约定，如上所述的那种死锁就可以避免。但是，下面的锁顺序规则说起来要比做起来容易。这类规则基本上不会出现在某个实际的系统中。通常，最好的办法是了解其他代码的做法。

有帮助的规则有两个。如果我们必须获得一个局部锁（比如一个设备锁），以及一个属于内核更中心位置的锁，则应该首先获取自己的局部锁。如果我们拥有信号量和自旋锁的组合，则必须首先获得信号量；在拥有自旋锁时调用*down*（可导致休眠）是个严重的错误的。当然，最好的办法是避免出现需要多个锁的情况。

细粒度锁和粗粒度锁的对比

第一个支持多处理器系统的Linux内核是2.0，其中有且只有一个锁。这个大的内核锁会让整个内核进入一个大的临界区，而只有一个CPU可以在任意给定时间执行内核代码。这种锁机制足以解决并发问题，从而允许内核开发者解决那些在支持SMP时遇到的所有问题。但是这种机制并不具有良好的伸缩性。即使是只有两个处理器的系统，也要在等待大的内核锁时花费大量时间。具有四个处理器的系统的性能甚至远远比不上四台独立的机器。

因此，其后的内核版本包含了更细粒度的锁。在2.2中，一个自旋锁控制对块I/O子系统的访问，而其他的自旋锁用于网络。现代的内核可包含数千个锁，每个锁保护一个小的资源。这种类型的细粒度锁具有良好的伸缩性；它允许每个处理器在执行特定任务时无需和其他处理器正在使用的锁竞争。因此，很少有人会想念早期的大内核锁（注3）。

注3：尽管目前这个锁在内核中已很少使用，但仍然存在于2.6中。如果读者不小心使用了*lock_kernel*调用，就会看到这个大的内核锁。但我们不应该在新代码中使用这个锁。

然而，细粒度锁本身有其成本。在包含数千个锁的内核中，为了执行某个特定的操作，我们很难确切知道需要锁定哪些锁，以及以什么样的顺序锁上它们。而锁定相关的缺陷又很难发现，这样，更多的锁导致锁缺陷在内核中危险蔓延的机会大大增加。细粒度的锁将带来某种程度的复杂性，并且随着时间的流逝，对内核的可维护性产生了很大的副作用。

设备驱动程序中的锁通常相对直接，我们可以用单个锁来处理所有的事情，或者可以为我们管理的每个设备建立一个锁。作为通常的规则，我们应该在最初使用粗粒度的锁，除非有真正的原因相信竞争会导致问题。我们需要抑制自己过早考虑优化的欲望，因为真正的性能约束通常出现在非预期的情况下。

如果我们的确怀疑锁竞争导致性能下降，则可以使用 *lockmeter* 工具。这个补丁（可在 *http://oss.sgi.com/projects/lockmeter/* 找到）可度量内核花费在锁上的时间。通过查看它的输出报告，我们可以很快确定锁竞争是否是问题所在。

除了锁之外的办法

Linux内核提供了大量有用的锁原语，它们却让内核步履蹒跚。但是如我们所看到的，锁机制的设计和实现本身并没有缺陷。通常，除了信号量或者自旋锁外我们别无选择，因为它们是完成某些工作的唯一选择。但是在某些情形下，原子的访问可以不需要完整的锁。本节将讨论不使用锁的方法。

免锁算法

有些时候，我们可以重新构造算法，以从根本上避免使用锁。大量的读取者/写入者情况——如果只有一个写入者——就可以用这种方法来设计我们的算法。如果写入者看到的数据结构和读取者看到的始终一致，就有可能构造一种免锁的数据结构。

经常用于免锁的生产者/消费者任务的数据结构之一是循环缓冲区（circular buffer）。在这个算法中，一个生产者将数据放入数组的结尾，而消费者从数组的另一端移走数据。在达到数组尾部的时候，生产者绕回到数组的头部。因此，一个循环缓冲区需要一个数组以及两个索引值，一个用于下一个要写入新值的位置，而另一个用于应下一个从缓冲区中移走值的位置。

如果仔细实现，在没有多个生产者或消费者的情况下，循环缓冲区不需要锁。生产者是唯一允许修改写入索引以及该索引指向的数组位置的线程。只要写入者在更新写入索引之前将新的值保存到缓冲区，则读取者将始终看到一致的数据结构。同时，读取者是唯

一可访问读取索引以及该索引指向位置的数据的线程。只要小心地确保两个指针不要互相重叠，生产者和消费者可以在没有竞态的情况下访问该缓冲区。

图5-1表示了填充循环缓冲区时的多个状态。当读取和写入指针相等时，表明缓冲区是空的，而只要写入指针马上要跑到读取指针的后面时（需谨慎处理交换！），就表明缓冲区已满。仔细编程，就可以在没有锁的情况下使用该缓冲区。

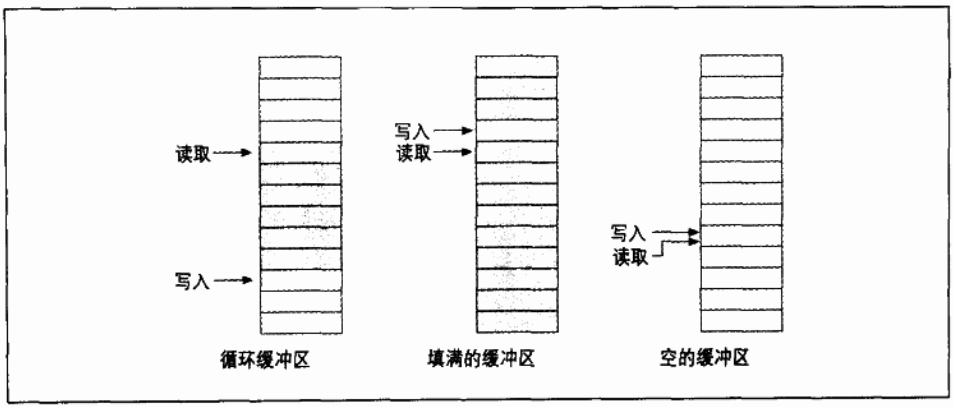


图5-1：循环缓冲区

循环缓冲区的使用在设备驱动程序中相当普遍。特别是网络适配器，经常使用循环缓冲区和处理器交换数据（数据包）。注意，在2.6.10中，内核有一个通用的循环缓冲区实现，有关其使用可参阅*<linux/kfifo.h>*。

原子变量

有时，共享的资源可能恰好是一个简单的整数值。假定我们的驱动程序维护着一个共享变量n_op，该变量的值表明有多少个设备操作正在并发地执行。通常，即使下面的简单操作也需要锁定：

```
n_op++;
```

某些处理器可以以原子的方式执行这类增加，但我们不能指望它。但话又说回来，完整的锁机制对一个简单的整数来讲却显得有些浪费。针对这种情况，内核提供了一种原子的整数类型，称为atomic_t，定义在*<asm/atomic.h>*中。

一个atomic_t变量在所有内核支持的架构上保存一个int值。但是，由于某些处理器上这种数据类型的工作方式有些限制，因此不能使用完整的整数范围；也就是说，在atomic_t变量中不能记录大于24位的整数。下面针对这种类型的操作在SMP计算机的

所有处理器上都确保是原子的。这种操作的速度非常快，因为只要可能，它们就会被编译成单个机器指令。

```
void atomic_set(atomic_t *v, int i);
atomic_t v = ATOMIC_INIT(0);
```

将原子变量v的值设置为整数值i。也可以在编译时利用ATOMIC_INIT宏来初始化原子变量的值。

```
int atomic_read(atomic_t *v);
```

返回v的当前值。

```
void atomic_add(int i, atomic_t *v);
```

将i累加到v指向的原子变量。返回值是void，这是因为返回新的值将带来额外的成本，而大多数情况下没有必要知道累加后的值。

```
void atomic_sub(int i, atomic_t *v);
```

从*v中减去i。

```
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
```

增加或缩减一个原子变量。

```
int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);
```

执行特定的操作并测试结果；如果在操作结束后，原子值为0，则返回值为true；否则返回值为false。注意，不存在*atomic_add_and_test*函数。

```
int atomic_add_negative(int i, atomic_t *v);
```

将整数变量i累加到v。返回值在结果为负时为true，否则为false。

```
int atomic_add_return(int i, atomic_t *v);
int atomic_sub_return(int i, atomic_t *v);
int atomic_inc_return(atomic_t *v);
int atomic_dec_return(atomic_t *v);
```

类似于*atomic_add*及其变种，例外之处在于这些函数会将新的值返回给调用者。

先前说过，atomic_t数据项必须只能通过上述函数来访问。如果读者将原子变量传递给了需要整型参数的函数，则会遇到编译错误。

还要记住，只有原子变量的数目是原子的，atomic_t变量才能工作。需要多个atomic_t变量的操作，仍然需要某种类型的锁。考虑下面的代码：

```
atomic_sub(amount, &first_atomic);
atomic_add(amount, &second_atomic);
```

在amount已经从第一个原子值中减去，到还没有增加到第二个原子值之间，会有一小段时间。如果可能在这两个操作之间运行的代码会导致问题的发生，则必须使用某种形式的锁。

位操作

atomic_t类型对执行整数算术来讲比较有用。但是当需要以原子形式来操作单个的位时，这种类型就无法派上用场了。为了实现位操作，内核提供了一组可原子地修改和测试单个位的函数。因为整个操作发生在单个步骤中，因此，不会受到中断（或者其他处理器）的干扰。

原子位操作非常快，只要底层硬件允许，这种操作就可以使用单个机器指令来执行，并且不需要禁止中断。这些函数依赖于具体的架构，因此在*<asm/bitops.h>*中声明。即使是在SMP计算机上，这些函数均可确保为原子的，因此能提供跨处理器的一致性。

不幸的是，这些函数使用的数据类型也是依赖于具体架构的。nr参数（用来描述要操作的位）通常被定义为int，但在少数架构上被定义为unsigned long。要修改的地址通常是指向unsigned long的指针，但在某些架构上却使用void *来代替。

可用的位操作如下：

```
void set_bit(nr, void *addr);
```

设置addr指向的数据项的第nr位。

```
void clear_bit(nr, void *addr);
```

清除addr指向的数据项的第nr位，其原语和*set_bit*相反。

```
void change_bit(nr, void *addr);
```

切换指定的位。

```
test_bit(nr, void *addr);
```

该函数是唯一一个不必以原子方式实现的位操作函数，它仅仅返回指定位的当前值。

```
int test_and_set_bit(nr, void *addr);
int test_and_clear_bit(nr, void *addr);
int test_and_change_bit(nr, void *addr);
```

像前面列出的函数一样具有原子化的行为，例外之处是它同时返回这个位的先前值。

当这些函数用来访问和修改一个共享标志时，除了调用它们之外，我们不需做其他任何事情 —— 它们会以原子方式执行操作。另一方面，使用位操作来管理一个锁变量以控制对某个共享变量的访问，则相对复杂并值得讨论。大多数现代的代码不会以这种方式使用位操作，但类似下面的代码仍在内核中存在。

要以原子方式获得锁并访问某个共享数据项的代码，可使用 *test_and_set_bit* 或者 *test_and_clear_bit*。常见的实现方法如下所列，该方法假定锁就是addr地址上的第nr位。它还假定当锁在零时空闲，而在非零时忙。

```
/* 试着设置锁定 */
while (test_and_set_bit(nr, addr) != 0)
    wait_for_a_while();

/* 完成自己的工作 */

/* 释放锁，并检查 */
if (test_and_clear_bit(nr, addr) =  = 0)
    something_went_wrong(); /* 已经被释放: 错误 */
```

如果读者通读内核源代码，将发现以上述方法工作的代码。然而，新代码应该使用自旋锁，因为自旋锁已被很好调试，并且能够处理类似中断和内核抢占这样的问题，而阅读你代码的其他人也不必花功夫来理解代码的意图。

seqlock

2.6内核包含有两个新的机制，可提供对共享资源的快速、免锁访问。当要保护的资源很小、很简单、会频繁被访问而且写入访问很少发生且必须快速时，就可以使用seqlock。从本质上讲，seqlock会允许读取者对资源的自由访问，但需要读取者检查是否和写入者发生冲突，当这种冲突发生时，就需要重试对资源的访问。seqlock通常不能用于保护包含有指针的数据结构，因为在写入者修改该数据结构的同时，读取者可能会追随一个无效的指针。

seqlock 在 *<linux/seqlock.h>* 中定义。通常用于初始化 seqlock（具有 seqlock_t 类型）的方法有如下两种：

```
seqlock_t lock1 = SEQLOCK_UNLOCKED;

seqlock_t lock2;
seqlock_init(&lock2);
```

读取访问通过获得一个（无符号的）整数顺序值而进入临界区。在退出时，该顺序值会和当前值比较；如果不相等，则必须重试读取访问。其结果是，读取者代码会如下编写：

```
unsigned int seq;
```

```
do {
    seq = read_seqbegin(&the_lock);
    /* 完成需要做的工作 */
} while read_seqretry(&the_lock, seq);
```

这种类型的锁通常用于保护某种类型的简单计算，这种计算需要多个一致的值。如果计算结束时发现已发生并发的修改，则可以简单丢弃结果并重新计算。

如果在中断处理例程中使用seqlock，则应该使用IRQ安全的版本：

```
unsigned int read_seqbegin_irqsave(seqlock_t *lock,
                                   unsigned long flags);
int read_seqretry_irqrestore(seqlock_t *lock, unsigned int seq,
                             unsigned long flags);
```

写入者必须在进入由seqlock保护的临界区时获得一个互斥锁。为此，需调用下面的函数：

```
void write_seqlock(seqlock_t *lock);
```

写入锁使用自旋锁实现，因此自旋锁的常见限制也适用于写入锁。做如下调用可释放该锁：

```
void write_sequnlock(seqlock_t *lock);
```

因为自旋锁用来控制写入访问，因此自旋锁的常见变种都可以使用，它们是：

```
void write_seqlock_irqsave(seqlock_t *lock, unsigned long flags);
void write_seqlock_irq(seqlock_t *lock);
void write_seqlock_bh(seqlock_t *lock);

void write_sequnlock_irqrestore(seqlock_t *lock, unsigned long flags);
void write_sequnlock_irq(seqlock_t *lock);
void write_sequnlock_bh(seqlock_t *lock);
```

如果*write_tryseqlock*可以获得自旋锁，它也会返回非零值。

读取－复制－更新

读取－复制－更新（read-copy-update，RCU）也是一种高级的互斥机制，在正确的条件下，也可获得高的性能。它很少在驱动程序中使用，但很知名，因此我们必须做一些基本的了解。对RCU算法的细节感兴趣的读者，可阅读由RCU的发明者发布的白皮书（*http://www.rdrop.com/users/paulmck/rclock/intro/rclock_intro.html*）。

RCU对它可以保护的数据结构做了一些限定。它针对经常发生读取而很少写入的情形做了优化。被保护的资源应该通过指针访问，而对这些资源的引用必须仅由原子代码拥有。

在需要修改该数据结构时，写入线程首先复制，然后修改副本，之后用新的版本替代相关指针，这也是该算法名称的由来。当内核确信老的版本上没有其他引用时，就可释放老的版本。

作为RCU的实际使用示例，可考虑网络路由表。每个外出数据包都需要检查路由表，以便确定应该使用哪个接口。这种检查很快，并且，一旦内核找到了目标接口，就不再需要那个路由表入口了。RCU可让路由查找无需锁定地实现，从而获得较大的性能提高。内核中的Starmode射频IP驱动程序也使用RCU来跟踪它自己的设备清单。

使用RCU的代码应包含*<linux/rcupdate.h>*。

在读取端，代码使用受RCU保护的数据结构时，必须将引用数据结构的代码包括在*rcu_read_lock*和*rcu_read_unlock*调用之间。这样，RCU代码可能如下所示：

```
struct my_stuff *stuff;

rcu_read_lock();
stuff = find_the_stuff(args...);
do_something_with(stuff);
rcu_read_unlock();
```

*rcu_read_lock*调用非常快，它会禁止内核抢占，但不会等待任何东西。用来检验读取“锁”的代码必须是原子的。在调用*rcu_read_unlock*之后，就不应该存在对受保护结构的任何引用。

用来修改受保护结构的代码必须在一个步骤中完成。第一步很简单，只需分配一个新的结构，如果必要则从老的结构中复制数据，然后将读取代码能看到的指针替换掉。这时，读取端会假定修改已经完成，任何进入临界区的代码将看到数据的新版本。

剩下的工作就是释放老的数据结构。当然，问题在于，在其他处理器上运行的代码可能仍在引用老的数据，因此不能立即释放老的结构。相反，写入代码必须等待直到能够确信不存在这样的引用。因为拥有对该数据结构的引用的代码都必须（规则决定）是原子的，因此我们可以知道，一旦系统中的每个处理器都至少调度一次之后，所有的引用都会消失。因此，RCU所做的就是，设置一个回调函数并等待所有的处理器被调度，之后由回调函数执行清除工作。

修改受RCU保护的数据结构的代码必须通过分配一个`struct rcu_head`数据结构来获得清除用的回调函数，但并不需要用什么方式来初始化这个结构。通常，这个结构内嵌在由RCU保护的大资源当中。在修改完资源之后，应该做如下调用：

```
void call_rcu(struct rcu_head *head, void (*func)(void *arg), void *arg);
```

在可安全释放该资源时，给定的 func 会被调用，传递到 *call_rcu* 的相同参数也会传递给这个函数。通常，func 要做的唯一工作就是调用 *kfree*。

完整的 RCU 接口要比我们看到的复杂得多，例如，它包括一些用来操作链表的辅助函数。读者可参阅相关头文件来了解这些接口。

快速参考

本章介绍了大量用来管理并发的符号，我们在这里总结了其中最重要的一些符号：

```
#include <asm/semaphore.h>
```

定义信号量及其操作的包含文件。

```
DECLARE_MUTEX(name);
DECLARE_MUTEX_LOCKED(name);
```

用于声明和初始化用在互斥模式中的信号量的两个宏。

```
void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(struct semaphore *sem);
```

这两个函数可在运行时初始化信号量。

```
void down(struct semaphore *sem);
int down_interruptible(struct semaphore *sem);
int down_trylock(struct semaphore *sem);
void up(struct semaphore *sem);
```

锁定和解锁信号量。如果必要，*down* 会将调用进程置于不可中断的休眠状态；相反，*down_interruptible* 可被信号中断。*down_trylock* 不会休眠，并且会在信号量不可用时立即返回。锁定信号量的代码最后必须使用 *up* 解锁该信号量。

```
struct rw_semaphore;
init_rwsem(struct rw_semaphore *sem);
```

信号量的读取者 / 写入者版本以及用来初始化这种信号量的函数。

```
void down_read(struct rw_semaphore *sem);
int down_read_trylock(struct rw_semaphore *sem);
void up_read(struct rw_semaphore *sem);
```

获取并释放对读取者 / 写入者信号量的读取访问的函数。

```
void down_write(struct rw_semaphore *sem)
int down_write_trylock(struct rw_semaphore *sem)
void up_write(struct rw_semaphore *sem)
void downgrade_write(struct rw_semaphore *sem)
```

对读取者/写入者信号量的写入访问进行管理的函数。

```
#include <linux/completion.h>
DECLARE_COMPLETION(name);
init_completion(struct completion *c);
INIT_COMPLETION(struct completion c);
```

描述 Linux 的 completion 机制的包含文件，以及用于初始化 completion 的常用方法。INIT_COMPLETION 只能用于对已使用过的 completion 的重新初始化。

```
void wait_for_completion(struct completion *c);
```

等待一个 completion 事件的发生。

```
void complete(struct completion *c);
void complete_all(struct completion *c);
```

发出completion事件信号。*complete*最多只能唤醒一个等待的线程，而*complete_all*会唤醒所有的等待者。

```
void complete_and_exit(struct completion *c, long retval);
```

通过调用 *complete* 并调用当前线程的 *exit* 函数而发出 completion 事件信号。

```
#include <linux/spinlock.h>
spinlock_t lock = SPIN_LOCK_UNLOCKED;
spin_lock_init(spinlock_t *lock);
```

定义自旋锁接口的包含文件，以及初始化自旋锁的两种方式。

```
void spin_lock(spinlock_t *lock);
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
void spin_lock_irq(spinlock_t *lock);
void spin_lock_bh(spinlock_t *lock);
```

锁定自旋锁的不同方式，某些方法会禁止中断。

```
int spin_trylock(spinlock_t *lock);
int spin_trylock_bh(spinlock_t *lock);
```

上述函数的非自旋版本。这些函数在无法获得自旋锁时返回零，否则返回非零。

```
void spin_unlock(spinlock_t *lock);
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
void spin_unlock_irq(spinlock_t *lock);
void spin_unlock_bh(spinlock_t *lock);
```

释放自旋锁的相应途径。

```
rwlock_t lock = RW_LOCK_UNLOCKED
rwlock_init(rwlock_t *lock);
```

初始化读取者/写入者锁的两种方式。

```
void read_lock(rwlock_t *lock);
void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
void read_lock_irq(rwlock_t *lock);
void read_lock_bh(rwlock_t *lock);
```

获取对读取者/写入者锁的读取访问的函数。

```
void read_unlock(rwlock_t *lock);
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void read_unlock_irq(rwlock_t *lock);
void read_unlock_bh(rwlock_t *lock);
```

释放对读取者/写入者自旋锁的读取访问的函数。

```
void write_lock(rwlock_t *lock);
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
void write_lock_irq(rwlock_t *lock);
void write_lock_bh(rwlock_t *lock);
```

获取对读取者/写入者自旋锁的写入访问的函数。

```
void write_unlock(rwlock_t *lock);
void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void write_unlock_irq(rwlock_t *lock);
void write_unlock_bh(rwlock_t *lock);
```

释放对读取者/写入者自旋锁的写入访问的函数。

```
#include <asm/atomic.h>
atomic_t v = ATOMIC_INIT(value);
void atomic_set(atomic_t *v, int i);
int atomic_read(atomic_t *v);
void atomic_add(int i, atomic_t *v);
void atomic_sub(int i, atomic_t *v);
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);
int atomic_add_negative(int i, atomic_t *v);
int atomic_add_return(int i, atomic_t *v);
int atomic_sub_return(int i, atomic_t *v);
int atomic_inc_return(atomic_t *v);
int atomic_dec_return(atomic_t *v);
```

整数变量的原子访问。对atomic_t变量的访问必须仅通过上述函数。

```
#include <asm/bitops.h>
void set_bit(nr, void *addr);
void clear_bit(nr, void *addr);
void change_bit(nr, void *addr);
test_bit(nr, void *addr);
int test_and_set_bit(nr, void *addr);
int test_and_clear_bit(nr, void *addr);
int test_and_change_bit(nr, void *addr);
```

对位值的原子访问，它们可用于标志或锁变量。使用这些函数可避免因为对相应位的并发访问而导致的任何竞态。

```
#include <linux/seqlock.h>
seqlock_t lock = SEQLOCK_UNLOCKED;
seqlock_init(seqlock_t *lock);
```

定义seqlock的包含文件，以及初始化seqlock的两种方式。

```
unsigned int read_seqbegin(seqlock_t *lock);
unsigned int read_seqbegin_irqsave(seqlock_t *lock, unsigned long flags);
int read_seqretry(seqlock_t *lock, unsigned int seq);
int read_seqretry_irqrestore(seqlock_t *lock, unsigned int seq, unsigned
  long flags);
```

用于获取受 seqlock 保护的资源的读取访问的函数。

```
void write_seqlock(seqlock_t *lock);
void write_seqlock_irqsave(seqlock_t *lock, unsigned long flags);
void write_seqlock_irq(seqlock_t *lock);
void write_seqlock_bh(seqlock_t *lock);
int write_tryseqlock(seqlock_t *lock);
```

用于获取受 seqlock 保护资源的写入访问的函数。

```
void write_sequnlock(seqlock_t *lock);
void write_sequnlock_irqrestore(seqlock_t *lock, unsigned long flags);
void write_sequnlock_irq(seqlock_t *lock);
void write_sequnlock_bh(seqlock_t *lock);
```

用于释放受 seqlock 保护的资源的写入访问的函数。

```
#include <linux/rcupdate.h>
```

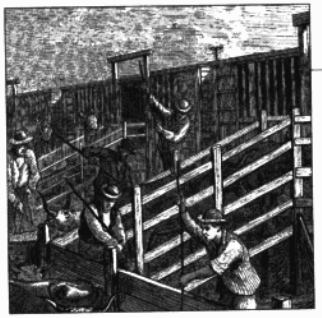
使用读取 - 复制 - 更新（RCU）机制时需要的包含文件。

```
void rcu_read_lock;
void rcu_read_unlock;
```

获取对受 RCU 保护的资源的读取访问的宏。

```
void call_rcu(struct rcu_head *head, void (*func)(void *arg), void *arg);
```

准备用于安全释放受 RCU 保护的资源的回调函数，该函数将在所有的处理器被调度后运行。



第六章

高级字符驱动程序操作

在第三章，我们已经构建了一个结构完整的可读写设备驱动程序。但一个实际可用的设备除了提供同步读取和写入之外，还会提供更多的功能。而现在我们拥有调试工具，掌握了相关的调试方法，并且对并发问题有了坚实的理解，这样，构造更高级的驱动程序就相对容易了。

本章阐述了编写全功能字符设备驱动程序的几个概念。首先，我们要实现 *ioctl* 系统调用，它是用于设备控制的公共接口。然后，我们介绍了和用户空间保持同步的几种途径。读完本章，读者将掌握如何使进程休眠（并唤醒）、如何实现非阻塞I/O，以及在设备可读取或写入时如何通知用户空间，等等。最后，我们介绍了如何在驱动程序中实现几种不同的设备访问策略。

本章介绍的这些内容均会通过对*scull*驱动程序的修改来说明，其实现使用的仍然是内存中的虚拟设备，这样，读者可以在不需要任何特定硬件的情况下尝试运行这些代码。读者也许着急处理真正的硬件，但还要等到第九章。

ioctl

除了读取和写入设备之外，大部分驱动程序还需要另外一种能力，即通过设备驱动程序执行各种类型的硬件控制。简单数据传输之外，大部分设备可以执行其他一些操作，比如，用户空间经常会请求设备锁门、弹出介质、报告错误信息、改变波特率或者执行自破坏，等等。这些操作通常通过 *ioctl* 方法支持，该方法实现了同名的系统调用。

在用户空间，*ioctl* 系统调用具有如下原型：

```
int ioctl(int fd, unsigned long cmd, ...);
```

由于使用了一连串的“.”的缘故，这个原型在Unix系统调用中显得比较特别，通常这

些点代表可变数目的参数表。但是在实际系统中，系统调用不会真正使用可变数目的参数，而是必须具有精确定义的原型，这是因为用户程序只能通过硬件“门”才能访问它们。所以，原型中的这些点并不是数目不定的一串参数，而只是一个可选参数，习惯上用char *argp定义。这里用点只是为了在编译时防止编译器进行类型检查。第三个参数的具体形式依赖于要完成的控制命令，也就是第二个参数。某些控制命令不需要参数，某些需要一个整数参数，而某些则需要一个指针参数。使用指针可以向*ioctl*调用传递任意数据，这样设备可以与用户空间交换任意数量的数据。

*ioctl*调用的非结构化本质导致众多内核开发者倾向于放弃它。从本质上讲，每个*ioctl*命令就是一个独立的系统调用，而且是非公开的，因此没有任何办法可以以一种容易理解的方式来审核这些调用。让这种非结构化的*ioctl*参数在所有系统上表现一致也是非常困难的，为了理解这一点，试想在64位系统上运行32位模式的用户空间进程。结果，有许多需求要求我们通过其他途径实现繁杂的控制操作，可能的方式包括：将命令嵌入到数据流中（将在本章后面讨论这一方法），或者使用虚拟文件系统，比如sysfs或者设备相关的文件系统（第十四章将讨论sysfs）。但现实情况中，对真正的设备操作来说，*ioctl*仍然是最简单且最直接的选择。

驱动程序的*ioctl*方法原型和用户空间的版本存在一些不同：

```
int (*ioctl) (struct inode *inode, struct file *filp,
              unsigned int cmd, unsigned long arg);
```

inode和filp两个指针的值对应于应用程序传递的文件描述符fd，这和传给*open*方法的参数一样。参数cmd由用户空间不经修改地传递给驱动程序，可选的arg参数则无论用户程序使用的是指针还是整数值，它都以unsigned long的形式传递给驱动程序。如果调用程序没有传递第三个参数，那么驱动程序所接收的arg参数就处在未定义状态。由于对这个附加参数的类型检查被关闭了，所以如果为*ioctl*传递一个非法参数，编译器是无法报警的，这样，相关联的程序错误就很难被发现。

读者可能已经想到了，大多数*ioctl*的实现中都包括一个switch语句来根据cmd参数选择对应的操作。不同的命令被赋予不同的数值，为了简化代码，通常会在代码中使用符号名代替数值，这些符号名由C语言的预处理语句定义。定制的设备驱动程序通常会在它们的头文件中声明这些符号，如*scull.h*中声明了*scull*所使用的符号。为了访问这些符号，用户程序自然也要包含这些头文件。

选择ioctl命令

在编写*ioctl*代码之前，需要选择对应不同命令的编号。多数程序员的第一本能是从0或者1开始选择一组小的编号。然而，有许多理由要求不能这样选择命令编号。为了防止

对错误的设备使用正确的命令，命令号应该在系统范围内唯一。这种错误匹配并不是不会发生，程序可能发现自己正在试图对FIFO和audio等这类非串行设备输入流修改波特率。如果每一个*ioctl*命令都是唯一的，应用程序进行这种操作时就会得到一个EINVAL错误，而不是无意间成功地完成了意想不到的操作。

为方便程序员创建唯一的*ioctl*命令号，每一个命令号被分为多个位字段。Linux的第一个版本使用了一个16位整数：高8位是与设备相关的“幻”数，低8位是一个序列号码，在设备内是唯一的。当时采用这种方案是因为，用Linus自己的话说，他有点“无头绪”，后来才得到一个更好的位字段分割方案。遗憾的是，相当多的驱动程序仍使用旧的约定，因为修改命令号会导致很多已有的二进制程序无法运行。

要按Linux内核的约定方法为驱动程序选择*ioctl*编号，应该首先看看*include/asm/ioctl.h*和*Documentation/ioctl-number.txt*这两个文件。头文件定义了要使用的位字段：类型（幻数）、序数、传送方向以及参数大小等等。*ioctl-number.txt*文件中罗列了内核所使用的幻数（注1），这样，在选择自己的幻数时就可以避免和内核冲突。这个文件也给出了为什么应该使用这个约定的原因。

定义号码的新方法使用了4个位字段，其含义如下面所给出。下面所介绍的新符号都定义在*<linux/ioctl.h>*中。

type
: 幻数。选择一个号码（记住先仔细阅读*ioctl-number.txt*），并在整个驱动程序中使用这个号码。这个字段有8位宽（_IOC_TYPEBITS）。

number
: 序数（顺序编号）。它也是8位宽（_IOC_NRBITS）。

direction
: 如果相关命令涉及到数据的传输，则该位字段定义数据传输的方向。可以使用的值包括_IOC_NONE（没有数据传输）、_IOC_READ、_IOC_WRITE以及_IOC_READ | _IOC_WRITE（双向传输数据）。数据传输是从应用程序的角度看的，也就是说，IOC_READ意味着从设备中读取数据，所以驱动程序必须向用户空间写入数据。注意，该字段是一个位掩码，因此可以用逻辑AND操作从中分解出_IOC_READ和_IOC_WRITE。

size
: 所涉及的用户数据大小。这个字段的宽度与体系结构有关，通常是13位或14位，具体可通过宏_IOC_SIZEBITS找到针对特定体系结构的具体数值。系统并不强制

注1：　但是，对这个文件的维护稍微有些滞后。

使用这个位字段，也就是说，内核不会检查这个位字段。对该位字段的正确使用可帮助我们检测用户空间程序的错误，并且如果我们从不改变相关数据项大小的话，这个位字段还可以帮助我们实现向后的兼容性。但是，如果需要很大的数据传输，则可以忽略这个位字段。稍后我们将介绍如何使用这个位字段。

<linux/ioctl.h> 中包含的 *<asm/ioctl.h>* 头文件定义了一些构造命令编号的宏：_IO(type,nr)用于构造无参数的命令编号；_IOR(type,nr,datatype)用于构造从驱动程序中读取数据的命令编号；_IOW(type,nr,datatype)用于写入数据的命令；_IOWR(type,nr,datatype)用于双向传输。type和number位字段通过参数传入，而size位字段通过对datatype参数取 *sizeof* 获得。

这个头文件还定义了用于解开位字段的宏：_IOC_DIR(nr)、_IOC_TYPE(nr)、_IOC_NR(nr)和_IOC_SIZE(nr)。在此不打算详细介绍这些宏，头文件里的定义已经足够清楚了，本节稍后也会给出示例。

下面是 *scull* 中的一些 *ioctl* 命令定义。需要特别指出的是，这些命令用来设置和获取驱动程序的配置参数。

```
/* 使用 "k" 作为幻数 */
#define SCULL_IOC_MAGIC  'k'
/* 在你自己的代码中，请使用不同的 8 位数字 */

#define SCULL_IOCRESET    _IO(SCULL_IOC_MAGIC, 0)

/*
 * S means "Set" through a ptr,
 * T means "Tell" directly with the argument value
 * G means "Get": reply by setting through a pointer
 * Q means "Query": response is on the return value
 * X means "eXchange": switch G and S atomically
 * H means "sHift": switch T and Q atomically
 */
/*
 * S 表示通过指针 "设置 (Set)"
 * T 表示直接用参数值 "通知 (Tell)"
 * G 表示 "获取 (Get)": 通过设置指针来应答
 * Q 表示 "查询 (Query)": 通过返回值应答
 * X 表示 "交换 (eXchange)": 原子地交换 G 和 S
 * H 表示 "切换 (sHift)": 原子地交换 T 和 Q
 */
#define SCULL_IOCSQUANTUM _IOW(SCULL_IOC_MAGIC,  1, int)
#define SCULL_IOCSQSET    _IOW(SCULL_IOC_MAGIC,  2, int)
#define SCULL_IOCTQUANTUM _IO(SCULL_IOC_MAGIC,   3)
#define SCULL_IOCTQSET    _IO(SCULL_IOC_MAGIC,   4)
#define SCULL_IOCGQUANTUM _IOR(SCULL_IOC_MAGIC,  5, int)
#define SCULL_IOCGQSET    _IOR(SCULL_IOC_MAGIC,  6, int)
#define SCULL_IOCQQUANTUM _IO(SCULL_IOC_MAGIC,   7)
#define SCULL_IOCQQSET    _IO(SCULL_IOC_MAGIC,   8)
```

```
#define SCULL_IOCXQUANTUM _IOWR(SCULL_IOC_MAGIC, 9, int)
#define SCULL_IOCXQSET    _IOWR(SCULL_IOC_MAGIC,10, int)
#define SCULL_IOCHQUANTUM _IO(SCULL_IOC_MAGIC,  11)
#define SCULL_IOCHQSET    _IO(SCULL_IOC_MAGIC,  12)

#define SCULL_IOC_MAXNR 14
```

实际的源码还定义了其他一些命令，但这里没有列出。

尽管根据已有的约定，*ioctl*应该使用指针完成数据交换，但我们仍然选择用两种方法实现整数参数传递 —— 通过指针和显式的数值。同样，这两种方法还用于返回整数：通过指针或通过设置返回值。如果返回值是正的，就表示工作正常。从任何一个系统调用返回时，正的返回值是受保护的（正如我们在*read*和*write*所见到的），而负值则被认为是一个错误，并被用来设置用户空间中的`errno`变量（注2）。

“exchange”和“shift”操作对*scull*设备来说并不特别有用。我们实现“exchange”操作是为了示范在驱动程序中如何把分离的操作合并成一个原子操作，而“shift”操作则将“tell”和“query”操作合并在一起。某些时候需要“测试兼设置”这类操作是原子操作 —— 特别是当应用程序需要加锁和解锁时。

显式的命令序数没什么特别含义，仅仅用来区分命令。其实，我们甚至可以在读命令和写命令中使用同一序数，因为实际*ioctl*编号中的“方向”位肯定不一样，不过最好还是不要这样做。除了在声明中用到序数之外，在别的地方我们都不用它，这样就不必为它分配一个符号了。这也就是为什么在前面给出的定义中直接使用了数字的原因。例子示范了一种使用命令编号的方法，读者也可以自行选择使用其他不同的方法。

除了少量预定义的命令（稍后讨论）之外，内核并未使用*ioctl*的`cmd`参数的值，以后也不太可能使用。这样，如果想偷懒，可以不使用上面那些复杂的声明，而直接显式地声明一组标量数字。由此带来的问题是，这样将无法从位字段中受益了。而且如果你打算将自己的代码合并到内核主线代码中的话，会带来许多问题。头文件*<linux/kd.h>*就是使用旧风格的例子，它使用了16位的标量数值来定义*ioctl*命令，这并非由于懒惰，而是那时只有这种方法，而现在修改它会引起一大堆兼容性方面的问题。

返回值

*ioctl*的实现通常就是一个基于命令号的`switch`语句。但是当命令号不能匹配任何合法的操作时，默认的选择是什么？对于这个问题颇有争议。有些内核函数会返回`-ENVAL`

注2：实际上，当前使用的所有*libc*实现（包括uClibc）认为错误范围在−4095 − −1之间。不幸的是，返回大的负错误号而不是小的错误号并不十分有用。

("Invalid argument，非法参数")，这是合理的，因为命令参数的确不是合法的参数。然而，POSIX标准规定，如果使用了不合适的*ioctl*命令参数，应该返回-ENOTTY。C库将这个错误码解释为"Inappropriate ioctl for device，不合适的设备ioctl"，这看起来更贴切些。尽管如此，对非法的*ioctl*命令返回-EINVAL仍然是很普遍的做法。

预定义命令

尽管*ioctl*系统调用绝大部分用于操作设备，但还有一些命令是可以由内核识别的。要注意，当这些命令用于我们的设备时，它们会在我们自己的文件操作被调用之前被解码。所以，如果你为自己的*ioctl*命令选用了与这些预定义命令相同的编号，就永远不会收到该命令的请求，而且由于*ioctl*编号冲突，应用程序的行为将无法预测。

预定义命令分为三组：

- 可用于任何文件（普通、设备、FIFO和套接字）的命令
- 只用于普通文件的命令
- 特定于文件系统类型的命令

最后一组命令只能在宿主文件系统上执行（见*chattr*命令）。设备驱动程序开发人员只对第一组感兴趣，它们的幻数都是"T"。分析其他组的工作留给读者做练习。*ext2_ioctl*是其中最有意思的函数（比读者想像的容易理解），它实现了只追加（append-only）标志和不可变（immutable）标志。

下列*ioctl*命令对任何文件（包括设备特定文件）都是预定义的：

FIOCLEX
: 设置执行时关闭标志（File IOctl CLose on EXec）。设置了这个标志之后，当调用进程执行一个新程序时，文件描述符将被关闭。

FIONCLEX
: 清除执行时关闭标志（File IOctl Not CLose on EXec）。该命令将恢复通常的文件行为，并撤销上述FIOCLEX命令所做的工作。

FIOASYNC
: 设置或复位文件异步通知（稍后在本章的"异步通知"一节中讨论）。注意，直到Linux 2.2.4版本的内核都不正确地使用了这个命令来修改O_SYNC标志。因为这两个动作都可以通过*fcntl*完成，所以实际上没有人会使用FIOASYNC命令，列在这里只是为了保持完整。

FIOQSIZE

该命令返回文件或目录的大小。不过，当用于设备文件时，会导致ENOTTY错误的返回。

FIONBIO

意指“File IOctl Non-Blocking I/O”，即“文件ioctl非阻塞型I/O”（稍后在本章“阻塞型与非阻塞型操作”一节中介绍）。该调用修改filp->f_flags中的O_NONBLOCK标志。传递给系统调用的第三个参数指明了是设置还是清除该标志。我们马上就可以看到该标志的作用。注意，修改这个标志的常用方法是由*fcntl*系统调用使用*F_SETFL*命令来完成。

在上述清单的最后一项中我们引入了一个新的系统调用，即*fcntl*，看起来很像*ioctl*。实际上，*fnctl*调用也要传递一个命令参数和一个附加的可选参数，在这点上它类似于*ioctl*。它和*ioctl*的不同主要是由于历史原因造成的：当Unix的开发人员面对控制I/O操作的问题时，他们认为文件和设备是不同的。那时，与*ioctl*实现相关的唯一设备就是终端，这也解释了为什么非法的*ioctl*命令的标准返回值是-ENOTTY。现在情况虽然不同了，但是*fcntl*还是为了向后兼容而保留了下来。

使用ioctl参数

在分析*scull*驱动程序的*ioctl*代码之前，还有一点要解释，就是怎样使用那个附加参数。如果它是个整数，那么很简单，直接使用就可以了。如果是个指针，就需要注意一些问题了。

当用一个指针指向用户空间时，必须确保指向的用户空间是合法的。对未验证的用户空间指针的访问，可能导致内核oops、系统崩溃或者安全问题。驱动程序应该负责对每个用到的用户空间地址做适当的检查，如果是非法地址则应该返回一个错误。

在第三章，我们看到了*copy_from_user*和*copy_to_user*函数，这两个函数可安全地与用户空间交换数据。这两个函数也可以在*ioctl*方法中使用，但是因为*ioctl*调用通常涉及到小的数据项，因此可通过其他方法更有效地操作。为此，我们首先要通过函数*access_ok*验证地址（而不传输数据），该函数在*<asm/uaccess.h>*中声明：

```
int access_ok(int type, const void *addr, unsigned long size);
```

第一个参数应该是VERIFY_READ或VERIFY_WRITE，取决于要执行的动作是读取还是写入用户空间内存区。addr参数是一个用户空间地址，size是字节数。例如，如果*ioctl*要从用户空间读取一个整数，size就是sizeof(int)。如果在指定地址处既要读取又要写入，则应该用VERIFY_WRITE，因为它是VERIFY_READ的超集。

与大多数函数不同，*access_ok* 返回一个布尔值：1 表示成功（访问成功），0 表示失败（访问不成功）。如果返回失败，驱动程序通常要返回 -EFAULT 给调用者。

关于 *access_ok*，有两点有趣之处需要注意。第一，它并没有完成验证内存的全部工作，而只检查了所引用的内存是否位于进程有对应访问权限的区域内，特别是要确保访问地址没有指向内核空间的内存区。第二，大多数驱动程序代码中都不需要真正调用 *access_ok*，因为后面要讲到的内存管理程序会处理它。尽管如此，我们还是示范一下它的使用。

scull 的源代码在 switch 语句前，通过分析 *ioctl* 编号的位字段来检查参数：

```
int err = 0, tmp;
int retval = 0;

/*
 * 抽取类型和编号位字段，并拒绝错误的命令号：
 * 在调用access_ok()之前返回 ENOTTY（不恰当的ioctl）
 */
 if (_IOC_TYPE(cmd) != SCULL_IOC_MAGIC) return -ENOTTY;
 if (_IOC_NR(cmd) > SCULL_IOC_MAXNR) return -ENOTTY;

/*
 * 方向是一个位掩码，而VERIFY_WRITE用于R/W* 传输。
 * “类型”是针对用户空间而言的，而access_ok是面向内核的。
 * 因此，“读取”和“写入”的概念恰好相反。
 */
if (_IOC_DIR(cmd) & _IOC_READ)
    err = !access_ok(VERIFY_WRITE, (void _ _user *)arg, _IOC_SIZE(cmd));
else if (_IOC_DIR(cmd) & _IOC_WRITE)
    err =  !access_ok(VERIFY_READ, (void _ _user *)arg, _IOC_SIZE(cmd));
if (err) return -EFAULT;
```

在调用 *access_ok* 之后，驱动程序就可以安全地进行实际的数据传送了。除了 *copy_from_user* 和 *copy_to_user* 函数外，程序员还可以使用已经为最常用的数据大小（1、2、4 及 8 个字节）优化过的一组函数。这些函数定义在 *<asm/uaccess.h>* 中，列在下面：

```
put_user(datum, ptr)
_ _put_user(datum, ptr)
```

这些宏把 datum 写到用户空间。它们相对比较快，当要传递单个数据时，应该用这些宏而不是用 *copy_to_user*。由于这些宏在展开时不做类型检查，所以可以传递给 *put_user* 任意类型的指针，只要是个用户空间地址就行。传递的数据大小依赖于 ptr 参数的类型，在编译时由编译器的内建指令 sizeof 和 typeof 确定。总之，如果 ptr 是一个字符指针，就传递 1 个字节、2、4、8 字节的情况类似。

*put_user*进行检查以确保进程可以写入指定的内存地址，并在成功时返回0，出错时返回-EFAULT。*__put_user*做的检查少些（它不调用*access_ok*），但如果地址指向用户不能写入的内存，也会出现操作失败。因而，*__put_user*应该在已经使用*access_ok*检验过内存区后再使用。

一般的用法是，实现一个读取方法时，可以调用*__put_user*来节省几个时钟周期，或者在复制多项数据之前调用一次*access_ok*，就像上面的*ioctl*代码一样。

`get_user(local, ptr)`
`__get_user(local, ptr)`

这些宏用于从用户空间接收一个数据。除了传输方向相反之外，它们与*put_user*和*__put_user*差不多。接收的数值被保存在局部变量`local`中，返回值则指明了操作是否成功。同样，*__get_user*应该在操作地址已被*access_ok*检验后使用。

如果试图使用上面列出的函数传递大小不符合任意一个特定值的数值，结果通常是编译器会给出一条奇怪的消息，比如“conversion to non-scalar type requested（需要转换为非标量类型）”。在这种情况下，必须使用*copy_to_user*或者*copy_from_user*。

权能与受限操作

对设备的访问由设备文件的权限控制，驱动程序通常不进行权限检查。不过也有这种情况，允许用户对设备读/写，而其他的操作被禁止。例如，不是所有的磁带驱动器使用者都可以设置它的默认块大小，允许用户使用磁盘设备也并不意味着就可以格式化磁盘。在类似的情况下，驱动程序必须进行附加的检查以确认用户是否有权进行请求的操作。

根据Unix系统的传统，特权操作仅限于超级用户账号。这种特权要么全有，要么全无——超级用户几乎可以做任何事，而所有其他用户则受到严格的限制。Linux内核提供了一个更为灵活的系统，称为权能（capability）。基于权能的系统抛弃了那种要么全有要么全无的特权分配方式，而是把特权操作划分为独立的组。这样，某个特定的用户或程序就可以被授权执行某一指定的特权操作，同时又没有执行其他不相关操作的能力。内核专为许可管理使用权能并导出了两个系统调用*capget*和*capset*，这样就可以从用户空间来管理权能。

全部权能操作都可以在*<linux/capability.h>*中找到，其中包含了系统能够理解的所有权能；不修改内核源代码，驱动程序作者或系统管理员就无法定义新的权能。对驱动程序开发者来讲有意义的权能如下所示：

`CAP_DAC_OVERRIDE`

越过文件或目录的访问限制（数据访问控制或DAC）的能力。

CAP_NET_ADMIN

执行网络管理任务的能力，包括那些能影响网络接口的任务。

CAP_SYS_MODULE

载入或卸除内核模块的能力。

CAP_SYS_RAWIO

执行“裸”I/O操作的能力。例如，访问设备端口或直接与USB设备通信。

CAP_SYS_ADMIN

截获的能力，它提供了访问许多系统管理操作的途径。

CAP_SYS_TTY_CONFIG

执行tty配置任务的能力。

在执行一项特权操作之前，设备驱动程序应该检查调用进程是否有合适的权能；如果不进行这类检查，将导致用户进程执行非授权操作，从而影响系统稳定性或安全性。权能的检查通过 *capable* 函数实现（定义在 *<sys/sched.h>* 中）：

```
int capable(int capability);
```

在 *scull* 示例驱动程序中，任何用户都被允许查询quantum和quantum集的大小。但是只有授权用户可以更改这些值，因为不恰当的值会降低系统性能。*scull* 的 *ioctl* 实现了在必要时检查用户的特权级别：

```
if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
```

因为缺少针对该任务的更多特定权能，所以这里使用了CAP_SYS_ADMIN。

ioctl命令的实现

scull 的 *ioctl* 实现中只传递设备的可配置参数，因此看起来很简单：

```
switch(cmd) {

  case SCULL_IOCRESET:
    scull_quantum = SCULL_QUANTUM;
    scull_qset = SCULL_QSET;
    break;

  case SCULL_IOCSQUANTUM: /* Set: arg指向参数值 */
    if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
    retval = __get_user(scull_quantum, (int __user *)arg);
    break;
```

```
  case SCULL_IOCTQUANTUM: /* Tell: arg本身就是参数值 */
    if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
    scull_quantum = arg;
    break;

  case SCULL_IOCGQUANTUM: /* Get: arg 是指向结果的指针 */
    retval = _ _put_user(scull_quantum, (int _ _user *)arg);
    break;

  case SCULL_IOCQQUANTUM: /* Query: 返回结果(结果是正值) */
    return scull_quantum;

  case SCULL_IOCXQUANTUM: /* eXchange: 将arg作为指针使用 */
    if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
    tmp = scull_quantum;
    retval = _ _get_user(scull_quantum, (int _ _user *)arg);
    if (retval =  = 0)
        retval = _ _put_user(tmp, (int _ _user *)arg);
    break;

  case SCULL_IOCHQUANTUM: /* sHift: 和Tell + Query类似 */
    if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
    tmp = scull_quantum;
    scull_quantum = arg;
    return tmp;

  default:  /* 冗余，因为cmd已根据MAXNR检查过了 */
    return -ENOTTY;
}
return retval;
```

*scull*中还包括6个操作scull_qset的入口，它们和scull_quantum的相应入口是一样的，这里不再赘述。

从调用方的观点（例如从用户空间）看，传送和接收参数的6种途径如下：

```
int quantum;

ioctl(fd,SCULL_IOCSQUANTUM, &quantum);          /* 通过指针设置 */
ioctl(fd,SCULL_IOCTQUANTUM, quantum);           /* 通过值设置 */

ioctl(fd,SCULL_IOCGQUANTUM, &quantum);          /* 通过指针获取 */
quantum = ioctl(fd,SCULL_IOCQQUANTUM);          /* 通过返回值获取 */

ioctl(fd,SCULL_IOCXQUANTUM, &quantum);          /* 通过指针交换 */
quantum = ioctl(fd,SCULL_IOCHQUANTUM, quantum); /* 通过值交换 */
```

当然，正常的驱动程序不会混用多种调用模式，在这里只是为了示范各种不同的方法。不过，通常情况下数据交换形式应该保持一致，要么都用指针，要么都用数值，尽量避免混用。

非 ioctl 的设备控制

有时通过向设备写入控制序列可以更好地控制设备。在控制台驱动程序中就使用了这一技术，称为“转义序列（escape sequence）”，用于控制移动光标、改变默认颜色或执行其他的配置任务。用这种方法实现设备控制的好处是，用户仅通过写数据就可以控制设备，无需使用（有时还得编写）配置设备的程序。如果我们用这种方式控制设备，发出命令的程序甚至无需运行在设备所在的同一系统上。

例如，*setterm* 程序通过打印转义序列来配置控制台（或某个终端）。控制程序可以运行在非被控设备所在的计算机上，然后用一个简单的数据流重定向就可以完成配置工作。每次我们运行一个远程的 tty 会话时，就会发生这种情况：转义序列从远程打印，而影响的却是本地 tty；当然，这一技术不仅仅限于 tty 设备。

通过打印序列进行控制的缺点是，它给设备增加了策略限制。例如，只有确定控制序列不会出现在写入设备的正常数据中时，才能使用这种技术。对 tty 设备来讲，只能部分满足这个要求。尽管文本显示设备的用途是显示ASCII字符，但有时写入数据流中也会出现控制字符，从而影响控制台的设置。例如，对一个二进制文件使用 *cat* 命令时，因为输出可能包含任何字符，其结果是经常会造成控制台的字体错误。

通过写入来控制的方式非常适合于那种不传送数据而只响应命令的设备，如机器人。

例如，笔者编写过一个驱动程序，该驱动程序控制相机在两个轴上移动。在这个驱动程序里，“设备”只是一对老式的步进马达，不能读写。“发送数据流”的概念对步进马达来说没什么意义。在这种情况下，驱动程序将所写的数据解释为ASCII命令，并把请求转换为脉冲序列来操纵步进马达。这种思路，与给调制解调器发送 AT 指令以设置通信的方法基本类似，主要区别在于连接调制解调器的串口还要发送真正的数据。直接设备控制的优点是使用 *cat* 就可以移动相机，而不必编写和编译用于实现 *ioctl* 调用的代码。

当编写这种“面向命令的”驱动程序时，没什么必要实现 *ioctl* 方法。在解释器中新增一条指令，其实现和使用都更简单。

尽管如此，有时可能需要做相反的事情：不是用 *write* 解释器来避免使用 *ioctl*，而是只使用 *ioctl*，完全不使用 *write*。同时，驱动程序附带了一个特定的命令行工具，专门负责把命令送给驱动程序。这种方法把内核空间的复杂性转移到了用户空间，这样处理起来可能会容易些，并且有助于缩小驱动程序的规模，然而，用户却无法再使用简单的命令（如 *cat* 或 *echo*）来操作驱动程序。

阻塞型 I/O

在第三章中，我们讨论了如何实现驱动程序的*read*和*write*方法。现在我们讨论另一个重要问题：如果驱动程序无法立即满足请求，该如何响应？当数据不可用时，用户可能调用*read*；或者进程试图写入数据，但因为输出缓冲区已满，设备还未准备好接受数据。调用进程通常不会关心这类问题，程序员只会简单调用*read*和*write*，然后等待必要的工作结束后返回调用。因此，在这种情况下，我们的驱动程序应该（默认）阻塞该进程，将其置入休眠状态直到请求可继续。

这一小节说明了如何使进程进入休眠并在将来唤醒。不过，我们首先要解释一些新的概念。

休眠的简单介绍

“休眠（sleep）”对进程来讲意味着什么？当一个进程被置入休眠时，它会被标记为一种特殊状态并从调度器的运行队列中移走。直到某些情况下修改了这个状态，进程才会在任意CPU上调度，也即运行该进程。休眠中的进程会被搁置在一边，等待将来的某个事件发生。

对Linux设备驱动程序来讲，让一个进程进入休眠状态很容易。但是，为了将进程以一种安全的方式进入休眠，我们需要牢记两条规则。

第一条规则是：永远不要在原子上下文中进入休眠。我们已经在第五章介绍过原子操作，而原子上下文就是指下面这种状态：在执行多个步骤时，不能有任何的并发访问。这意味着，对休眠来说，我们的驱动程序不能在拥有自旋锁、seqlock或者RCU锁时休眠。如果我们已经禁止了中断，也不能休眠。在拥有信号量时休眠是合法的，但是必须仔细检查拥有信号量时休眠的代码。如果代码在拥有信号量时休眠，任何其他等待该信号量的线程也会休眠，因此任何拥有信号量而休眠的代码必须很短，并且还要确保拥有信号量并不会阻塞最终会唤醒我们自己的那个进程。

另外一个需要铭记的是：当我们被唤醒时，我们永远无法知道休眠了多长时间，或者休眠期间都发生了些什么事情。我们通常也无法知道是否还有其他进程在同一事件上休眠，这个进程可能会在我们之前被唤醒并将我们等待的资源拿走。这样，我们对唤醒之后的状态不能做任何假定，因此必须检查以确保我们等待的条件真正为真。

另外一个相关的问题是，除非我们知道有其他人会在其他地方唤醒我们，否则进程不能休眠。完成唤醒任务的代码还必须能够找到我们的进程，这样才能唤醒休眠的进程。为确保唤醒发生，需整体理解我们的代码，并清楚地知道对每个休眠而言哪些事件序列会

结束休眠。能够找到休眠的进程意味着，需要维护一个称为等待队列的数据结构。顾名思义，等待队列就是一个进程链表，其中包含了等待某个特定事件的所有进程。

在Linux中，一个等待队列通过一个“等待队列头（wait queue head）”来管理，等待队列头是一个类型为wait_queue_head_t的结构体，定义在*<linux/wait.h>*中。可通过如下方法静态定义并初始化一个等待队列头：

```
DECLARE_WAIT_QUEUE_HEAD(name);
```

或者使用动态方法：

```
wait_queue_head_t my_queue;
init_waitqueue_head(&my_queue);
```

稍后我们将继续解释等待队列结构，但现在还需要继续讨论休眠和唤醒。

简单休眠

当进程休眠时，它将期待某个条件会在未来成为真。我们前面提到，当一个休眠进程被唤醒时，它必须再次检查它所等待的条件的确为真。Linux内核中最简单的休眠方式是称为*wait_event*的宏（以及它的几个变种）；在实现休眠的同时，它也检查进程等待的条件。*wait_event*的形式如下：

```
wait_event(queue, condition)
wait_event_interruptible(queue, condition)
wait_event_timeout(queue, condition, timeout)
wait_event_interruptible_timeout(queue, condition, timeout)
```

在上面所有的形式中，queue是等待队列头。注意，它“通过值”传递，而不是通过指针。condition是任意一个布尔表达式，上面的宏在休眠前后都要对该表达式求值；在条件为真之前，进程会保持休眠。注意，该条件可能会被多次求值，因此对该表达式的求值不能带来任何副作用。

如果使用*wait_event*，进程将被置于非中断休眠，如我们先前提到的，这通常不是我们所期望的。最好的选择是使用*wait_event_interruptible*，它可以被信号中断。这个版本可返回一个整数值，非零值表示休眠被某个信号中断，而驱动程序也许要返回-ERESTARTSYS。后面的两个版本（*wait_event_timeout*和*wait_event_interruptible_timeout*）只会等待限定的时间；当给定的时间（以jiffy表示，第七章将讨论）到期时，这两个宏都会返回0值，而无论condition如何求值。

当然，整个过程的另外一半是唤醒。其他的某个执行线程（可能是另一个进程或者中断处理例程）必须为我们执行唤醒，因为我们的进程正在休眠中。用来唤醒休眠进程的基本函数是*wake_up*，它也有多种形式，但这里先介绍其中两个：

```
void wake_up(wait_queue_head_t *queue);
void wake_up_interruptible(wait_queue_head_t *queue);
```

*wake_up*会唤醒等待在给定queue上的所有进程（实际情况要复杂一些，读者很快会看到）。另一个形式（*wake_up_interruptible*）只会唤醒那些执行可中断休眠的进程。通常，这两种形式很难区分（如果使用可中断休眠的话）；在实践中，约定作法是在使用*wait_event*时使用*wake_up*，而在使用*wait_event_interruptible*时使用*wake_up_interruptible*。

现在我们看看休眠和唤醒的一个简单例子。在示例源代码中，读者可以找到一个称为*sleepy*的模块，它实现了一个具有简单行为的设备：任何试图从该设备上读取的进程均被置于休眠。只要某个进程向该设备写入，所有休眠的进程就会被唤醒。这一行为通过下面的*read*和*write*方法实现：

```
static DECLARE_WAIT_QUEUE_HEAD(wq);
static int flag = 0;

ssize_t sleepy_read (struct file *filp, char _ _user *buf, size_t count,
loff_t *pos)
{
    printk(KERN_DEBUG "process %i (%s) going to sleep\n",
           current->pid, current->comm);
    wait_event_interruptible(wq, flag != 0);
    flag = 0;
    printk(KERN_DEBUG "awoken %i (%s)\n", current->pid, current->comm);
    return 0; /* EOF */
}

ssize_t sleepy_write (struct file *filp, const char _ _user *buf, size_t
count,
        loff_t *pos)
{
    printk(KERN_DEBUG "process %i (%s) awakening the readers...\n",
           current->pid, current->comm);
    flag = 1;
    wake_up_interruptible(&wq);
    return count; /* 成功并避免重试 */
}
```

注意上面例子中flag变量的使用。因为*wait_event_interrutible*要检查改变为真的条件，因此我们使用flag来构造这个条件。

请读者想像当两个进程在等待时调用*sleepy_write*，会发生什么情况。因为*sleepy_read*会在唤醒时将flag重置为0，因此，读者会认为第二个被唤醒的进程会立即进入休眠状态。在单处理器系统中，这几乎是始终会发生的情况。然而，我们必须理解并不是永远会发生这种情况。*wake_up_interruptible*调用会唤醒两个休眠的进程，而在重置flag之前，这两个进程都完全有可能注意到标志为非零。对这个不重要的模块来讲，这种竞态并不重要。但在真实的驱动程序中，这种类型的竞态可能导致很难诊断的、偶然的崩

溃。如果要确保只有一个进程能看到非零值，则必须以原子方式进行检查。我们将很快看到真实的驱动程序如何处理这类问题，但首先要讨论另外一个主题。

阻塞和非阻塞型操作

在分析用于休眠进程的、功能完整的*read*和*write*方法之前，还有最后一个问题需要讨论。在实现正确的Unix语义时，有时我们要实现非阻塞的操作，尽管操作不能完整执行。

有时调用进程会通知我们它不想阻塞，而不管其I/O是否可以继续。显式的非阻塞I/O由`filp->f_flags`中的`O_NONBLOCK`标志决定。这个标志在*<linux/fcntl.h>*中定义，这个头文件自动包含在*<linux/fs.h>*中。这个标志的名字取自“非阻塞打开（open-nonblock）”，因为它可以在打开时指定（而且本来也只能在那时指定）。浏览一下源代码，会发现一些对`O_NDELAY`标志的引用，这是`O_NONBLOCK`的另一个名字，是为保持和System V代码的兼容性而设计的。这个标志在默认时要被清除，因为等待数据的进程一般只是休眠。在执行阻塞型操作（这是默认的）的情况下，应该实现下列动作以保持和标准语义一致：

- 如果一个进程调用了*read*但是还没有数据可读，此进程必须阻塞。数据到达时进程被唤醒，并把数据返回给调用者。即使数据数目少于`count`参数指定的数目也是如此。
- 如果一个进程调用了*write*但缓冲区没有空间，此进程必须阻塞，而且必须休眠在与读取进程不同的等待队列上。当向硬件设备写入一些数据，从而腾出了部分输出缓冲区后，进程即被唤醒，*write*调用成功。即使缓冲区中可能没有所要求的`count`字节的空间而只写入了部分数据，也是如此。

上面的描述假设输入和输出缓冲区都存在，实际上它们也确实存在于绝大多数设备中。输入缓冲区用于当数据已到达而又无人读取时，把数据暂存起来避免丢失；相反，如果调用*write*时系统不能接收数据，就将它们保留在用户空间缓冲区中不致会丢失。除此以外，输出缓冲区几乎总是可以提高硬件的性能。

在驱动程序中实现输出缓冲区可以提高性能，这得益于减少了上下文切换和用户级/内核级转换的次数。假设一个慢速设备没有输出缓冲区，那么每次系统调用只能接收一个或几个字符，然后进程在*write*上休眠，另一个进程开始运行（这里有一次上下文切换），当前一个进程被唤醒后，它重新开始运行（引起另一次上下文切换），*write*返回（内核/用户转换），接着进程重复系统调用写入更多数据（用户/内核转换），接着调用又阻塞，然后再次进行以上的循环。如果输出缓冲区足够大，那么*write*调用首次操作就成功了——缓存的数据可以在以后的中断时间送给设备——而不必返回用户空间为第二次或第三次的*write*调用进行控制。显然，输出缓冲区多大才合适是与设备相关的。

在*scull*中没有使用输入缓冲区，因为调用*read*时，数据已经就绪了。类似地，也没有输出缓冲区，因为数据只是简单地被复制到与设备对应的内存区。其实该设备本身就是一个缓冲区，因此不必实现另外的缓冲区。我们将在第十章介绍缓冲区的使用。

如果指定了O_NONBLOCK标志，*read*和*write*的行为就会有所不同。如果在数据没有就绪时调用*read*或是在缓冲区没有空间时调用*write*，则该调用简单地返回-EAGAIN。

读者可能已经想到，非阻塞型操作会立即返回，使得应用程序可以查询数据。在处理非阻塞型文件时，应用程序调用*stdio*函数必须非常小心，因为很容易把一个非阻塞返回误认为是EOF，所以必须始终检查errno。

自然，O_NONBLOCK在*open*方法中也是有意义的。它用于在*open*调用可能会阻塞很长时间的场合。例如，打开一个还没有进程向其中写入的FIFO或是访问一个被锁住的磁盘文件。通常情况下，打开一个设备不是成功就是失败，不必等待外部事件。但是有时候打开设备需要很长时间的初始化，这时就可以选择在*open*方法中支持O_NONBLOCK标志，如果该标志被置位，则在设备开始初始化后会立刻返回一个-EAGAIN（“try it again，再试一次”）。驱动程序中也可以实现阻塞型*open*以支持和文件锁方式类似的访问策略。在本章的“替代EBUSY的阻塞型open”一节中就会看到这样一个实现。

有些驱动程序还为O_NONBLOCK实现了特殊的语义。例如，在磁带还没有插入时打开一个磁带设备通常会阻塞，如果磁带驱动程序是用O_NONBLOCK打开的，则不管磁带在不在，open都会立即成功返回。

只有*read*、*write*和*open*文件操作受非阻塞标志的影响。

一个阻塞I/O示例

最后，我们通过一个示例来分析实现阻塞I/O的真实驱动程序方法。这个例子来自*scullpipe*驱动程序，它是*scull*实现类管道设备的特殊形式。

在驱动程序内部，阻塞在*read*调用的进程在数据到达时被唤醒；通常硬件会发出一个中断来通知这个事件，然后作为中断处理的一部分，驱动程序会唤醒等待进程。*scullpipe*驱动程序的工作方法则不同，它不需要任何特殊的硬件或是中断处理程序就可以运行。我们选择使用另一个进程来产生数据并唤醒读取进程；类似地，读取进程用来唤醒等待缓冲区空间可用的写入进程。

该设备驱动程序使用了一个包含两个等待队列和一个缓冲区的设备结构。缓冲区大小可以用通常的方式配置（在编译、加载或运行时）。

```
struct scull_pipe {
        wait_queue_head_t inq, outq;        /* 读取和写入队列 */
        char *buffer, *end;                 /* 缓冲区的起始和结尾 */
        int buffersize;                     /* 用于指针计算 */
        char *rp, *wp;                      /* 读取和写入的位置 */
        int nreaders, nwriters;             /* 用于读写打开的数量*/
        struct fasync_struct *async_queue; /* 异步读取者 */
        struct semaphore sem;               /* 互斥信号量 */
        struct cdev cdev;                   /* 字符设备结构 */
};
```

read 实现负责管理阻塞型和非阻塞型输入，如下所示：

```
static ssize_t scull_p_read (struct file *filp, char _ _user *buf, size_t count,
                loff_t *f_pos)
{
    struct scull_pipe *dev = filp->private_data;

    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;

    while (dev->rp =  = dev->wp) { /* 无数据可读取 */
        up(&dev->sem); /* 释放锁 */
        if (filp->f_flags & O_NONBLOCK)
            return -EAGAIN;
        PDEBUG("\"%s\" reading: going to sleep\n", current->comm);
        if (wait_event_interruptible(dev->inq, (dev->rp != dev->wp)))
            return -ERESTARTSYS; /* 信号，通知 fs 层做相应处理 */
        /* 否则循环，但首先获取锁 */
        if (down_interruptible(&dev->sem))
            return -ERESTARTSYS;
    }
    /* 数据已就绪，返回*/
   if (dev->wp > dev->rp)
        count = min(count, (size_t)(dev->wp - dev->rp));
    else /* 写入指针回卷，返回数据直到 dev->end */
        count = min(count, (size_t)(dev->end - dev->rp));
    if (copy_to_user(buf, dev->rp, count)) {
        up (&dev->sem);
        return -EFAULT;
    }
    dev->rp += count;
    if (dev->rp =  = dev->end)
        dev->rp = dev->buffer; /* 回卷 */
    up (&dev->sem);

    /* 最后，唤醒所有写入者并返回 */
    wake_up_interruptible(&dev->outq);
    PDEBUG("\"%s\" did read %li bytes\n",current->comm, (long)count);
    return count;
}
```

可以看到代码中保留了一些PDEBUG语句。编译该驱动程序时可以启用消息以便跟踪不同进程间的交互。

现在仔细看看*scull_p_read*是如何处理等待数据的。while循环在拥有设备信号量时测试缓冲区。如果其中有数据，则可以立即将数据返回给用户而不需要休眠，这样，整个循环体就被跳过了。相反，如果缓冲区为空，则必须休眠。但在休眠之前必须释放设备信号量，因为如果在拥有该信号量时休眠，任何写入者都没有机会来唤醒。在释放信号量之后，快速检查用户请求的是否是非阻塞I/O，如果是，则返回，否则调用*wait_event_interruptible*。

在上面这个函数调用返回时，说明其他人已经唤醒了我们，但我们不知道到底是什么情况。一种可能性是进程接收到一个信号。包含*wait_event_interruptible*调用的if语句检查这种情况。这条语句确保对信号进行正确的预定响应，该信号可能是用来唤醒进程的（因为进程处于可中断睡眠中）。如果一个信号到达而且没有被进程阻塞，正确的动作是让内核的上层去处理这个事件。为此，驱动程序返回给调用者-ERESTARTSYS，这个值由虚拟文件系统层（VFS）内部使用，它或者重启系统调用，或者给用户空间返回-EINTR。我们将在所有的*read*和*write*实现中使用同样的语句进行信号处理。

但是，就算不是因为信号而被唤醒，我们还是无法确信是否有数据可获得。其他人可能也在等待数据，而且可能赢得竞争并拿走了数据。因此，我们必须重新获得设备信号量，只有这样才能测试读取缓冲区（在while循环中），并真正知道我们是否可以将缓冲区的数据返回给用户。整个代码的最终结果就是，当我们从while循环中退出时，我们拥有信号量，而且缓冲区中包含有可使用的数据。

出于完整性考虑，还要注意*scull_p_read*可能在我们获取设备信号量之后休眠的另外一种情况，即调用*copy_to_user*。如果*scull*在内核和用户空间复制数据时休眠，则会在拥有设备信号量时休眠。在这种情况下，拥有信号量是可以接受的，因为这不会死锁系统（我们知道内核会将数据复制到用户空间然后唤醒我们，同时不会试图锁上同一信号量），而且重要的是，在驱动程序休眠时，设备的内存数组不会被修改。

高级休眠

我们介绍过的函数可以满足许多驱动程序的休眠需求。但是在某些情况下，我们需要对Linux的等待队列机制有更加深入的理解。复杂的锁定以及性能需求会强制驱动程序使用低层的函数来实现休眠。本小节中，我们将讨论一些低层次的细节，以便理解进程休眠时到底发生了什么事情。

进程如何休眠

如果读者浏览<*linux/wait.h*>头文件，将看到wait_queue_head_t类型后面的数据结构相当简单，它由一个自旋锁和一个链表组成。链表中保存的是一个等待队列入口，该入口声明为wait_queue_t类型。这个结构中包含了休眠进程的信息及其期望被唤醒的相关细节信息。

将进程置于休眠的第一个步骤通常是分配并初始化一个wait_queue_t结构，然后将其加入到对应的等待队列。在完成这些工作之后，不管谁负责唤醒该进程，都能找到正确的进程。

第二个步骤是设置进程的状态，将其标记为休眠。<*linux/sched.h*>中定义了多个任务状态。TASK_RUNNING表示进程可运行，尽管进程并不一定在任何给定时间都运行在某个处理器上。有两个状态表明进程处于休眠状态：TASK_INTERRUPTIBLE和TASK_UNINTERRUPUTIBLE；显然，它们分别对应于两种休眠。其他的状态对驱动程序编写者来说通常不需要关心。

在2.6内核中，通常不需要驱动程序代码来直接操作进程状态。但是如果需要，则可调用：

```
void set_current_state(int new_state);
```

在老的代码中，读者通常可能会找到下面的语句：

```
current->state = TASK_INTERRUPTIBLE;
```

但是我们不鼓励以这种方式直接修改current，因为数据结构的改变很容易导致该代码无法运行，而且上述修改进程当前状态的代码并不会将自己置于休眠状态。通过改变当前状态，我们只是改变了调度器处理该进程的方式，但尚未使进程让出处理器。

放弃处理器是最后的步骤，但在此之前还要做另外一件事情：我们必须首先检查休眠等待的条件。如果不作这个检查，可能引入竞态。试想，如果在上述过程中条件变成了真，而其他线程正试图唤醒我们，这时会发生什么呢？我们会丢掉被唤醒的机会，从而可能休眠更长的时间。因此，深入到休眠的代码，我们会看到下面的语句：

```
if (!condition)
    schedule();
```

在设置了进程状态之后检查条件，我们解决了所有可能的事件序列。如果我们等待的条件在设置进程状态前发生，我们会在这个检查中注意且不会真正休眠。如果唤醒在其后发生，不管我们是否真正进入休眠，进程都会被置于可运行状态。

当然，对*schedule*的调用将调用调度器，并让出CPU。无论在什么时候调用这个函数，都将告诉内核重新选择其他进程运行，并在必要时将控制切换到那个进程。这样，我们无法知道，在调度返回到我们的代码之前需要多少时间。

在if测试以及可能的*schedule*调用（并返回）之后，需要完成一些清理工作。因为代码不再期望休眠，因此必须确保任务状态被重置为TASK_RUNNING。如果代码从*schedule*中返回，则不需要这一步。但是，如果因为不需要休眠而跳过了对*schedule*的调用，那么进程状态就是不正确的。将进程从等待队列中移走也是必要的，否则它可能会被多次唤醒。

手工休眠

在早期的Linux内核版本中，特殊休眠需要程序员手工处理上面讲过的所有步骤。这是一个冗长而乏味的过程，涉及到相当多的、容易导致错误的代码。如果开发者愿意，仍可以沿用这种手工休眠的方式。*<linux/sched.h>*中包含有所有必需的定义，而内核源代码中也含有大量的例子。但是，也有一种更加简单的方式。

第一个步骤是建立并初始化一个等待队列入口。这通常通过下面的宏完成：

```
DEFINE_WAIT(my_wait);
```

其中的name是等待队列入口变量的名称。我们也可以通过下面两个步骤完成这个工作：

```
wait_queue_t my_wait;
init_wait(&my_wait);
```

但是，在实现休眠的循环前放置DEFINE_WAIT行通常更加容易些。

下一个步骤是将我们的等待队列入口添加到队列中，并设置进程的状态。这两个任务都可通过下面的函数完成：

```
void prepare_to_wait(wait_queue_head_t *queue,
                     wait_queue_t *wait,
                     int state);
```

其中，queue和wait分别是等待队列头和进程入口。state是进程的新状态；它应该是TASK_INTERRUPTIBLE（用于可中断休眠，通常也是我们所期望的）或者TASK_UNINTERRUPTIBLE（用于不可中断休眠）。

在调用*prepare_to_wait*之后，进程即可调用*schedule*，当然在这之前，应确保仍有必要等待。一旦*schedule*返回，就到了清理时间了。这个工作也可通过下面的特殊函数完成：

```
void finish_wait(wait_queue_head_t *queue, wait_queue_t *wait);
```

之后，代码可测试其状态，并判断是否需要重新等待。

现在看看实际的例子。前面我们分析了*scullpipe*的*read*方法，其中使用了*wait_event*。但这个驱动程序的*write*方法使用了*prepare_to_wait*和*finish_wait*。通常，我们不应该在一个驱动程序中混合使用两种方法，但是这里只是希望说明处理休眠的两种方法而已。

为了完整，我们首先看看*write*方法本身：

```
/* 有多少空间被释放? */
static int spacefree(struct scull_pipe *dev)
{
    if (dev->rp == dev->wp)
        return dev->buffersize - 1;
    return ((dev->rp + dev->buffersize - dev->wp) % dev->buffersize) - 1;
}

static ssize_t scull_p_write(struct file *filp, const char __user *buf,
size_t count,
                loff_t *f_pos)
{
    struct scull_pipe *dev = filp->private_data;
    int result;

    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
    /* 确保有空间可写入 */
    result = scull_getwritespace(dev, filp);
    if (result)
        return result; /* scull_getwritespace会调用 up(&dev->sem) */

    /* 有空间可用，接受数据 */
    count = min(count, (size_t)spacefree(dev));
    if (dev->wp >= dev->rp)
        count = min(count, (size_t)(dev->end - dev->wp)); /* 直到缓冲区结尾 */
    else /* 写入指针回卷，填充到 rp-1 */
        count = min(count, (size_t)(dev->rp - dev->wp - 1));
    PDEBUG("Going to accept %li bytes to %p from %p\n", (long)count, dev->wp, buf);
    if (copy_from_user(dev->wp, buf, count)) {
        up (&dev->sem);
        return -EFAULT;
    }
    dev->wp += count;
    if (dev->wp == dev->end)
        dev->wp = dev->buffer; /* 回卷 */
    up(&dev->sem);

    /* 最后，唤醒读取者 */
    wake_up_interruptible(&dev->inq);  /* 阻塞在read()和select()上 */

    /* 通知异步读取者，将在本章后面解释 */
    if (dev->async_queue)
        kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
```

```
    PDEBUG("\"%s\" did write %li bytes\n",current->comm, (long)count);
    return count;
}
```

代码看起来和*read*方法非常相似，除了我们将休眠的代码放在了独立的*scull_getwritespace*函数中。该函数确保新数据有可用的缓冲区空间，并且在必要时休眠、直到空间可用。一旦获得缓冲区空间，*scull_p_write*即可将用户数据复制到其中，调整指针，并唤醒可能正在等待读取数据的任何进程。

处理真正休眠的代码如下：

```
/* 等待有可用于写入的空间；调用者必须拥有设备信号量。
 * 在错误情况下，信号量将在返回前释放。 */
static int scull_getwritespace(struct scull_pipe *dev, struct file *filp)
{
    while (spacefree(dev) ==  = 0) { /* full */
        DEFINE_WAIT(wait);

        up(&dev->sem);
        if (filp->f_flags & O_NONBLOCK)
            return -EAGAIN;
        PDEBUG("\"%s\" writing: going to sleep\n",current->comm);
        prepare_to_wait(&dev->outq, &wait, TASK_INTERRUPTIBLE);
        if (spacefree(dev) ==  = 0)
            schedule();
        finish_wait(&dev->outq, &wait);
        if (signal_pending(current))
            return -ERESTARTSYS; /* 信号：通知fs 层做相应处理 */
        if (down_interruptible(&dev->sem))
            return -ERESTARTSYS;
    }
    return 0;
}
```

再次注意其中的while循环。如果不需要休眠的情况下空间即可用，该函数立即返回。否则，该函数释放设备信号量并等待。代码使用了*DEFINE_WAIT*来设置等待队列入口，并调用*prepare_to_wait*来进入真正的休眠。之后对缓冲区做必要的检查——我们必须处理以下情况：在我们进入while循环（并释放信号量）之后，但在将自己放到等待队列之前，缓冲区空间变得可用。若不作这个检查，如果读取进程在此时完全清空了缓冲区，则我们将失去唯一被唤醒的机会，从而会永远休眠。在到达必须休眠的条件时，即可调用*schedule*。

对上述情况还值得再次考虑：如果唤醒发生在if测试以及对*schedule*的调用之间，会发生什么呢？在这种情况下，不会出现任何问题。休眠将会把进程状态置为TASK_RUNNING，而*schedule*会返回——尽管根本没必要调用*schedule*。只要测试发生在进程已经将自己放在等待队列并修改了其状态之后，就不会出现任何问题。

在最后，我们调用*finish_wait*。对*signal_pending*的调用告诉我们是否因为信号而被唤醒。如果是，我们需要返回给用户并让用户再次重试；否则，我们将获取信号量，并像通常那样再次测试空闲空间。

独占等待

我们已经看到，当某个进程在等待队列上调用*wake_up*时，所有等待在该队列上的进程都将被置为可运行状态。在许多情况下，这是正确的行为；但在其他情况下，我们可以预先知道只会有一个被唤醒的进程可以获得期望的资源，而其他被唤醒的进程只会再次休眠。这些被唤醒进程中的每一个都要获得处理器，为资源（以及锁）竞争，然后又再次进入休眠。如果等待队列中的进程数量非常庞大，这种“疯狂兽群”行为将严重影响系统性能。

为了解决现实世界中的“疯狂兽群”问题，内核开发者为内核增加了“独占等待”选项。一个独占等待的行为和通常的休眠类似，但有如下两个重要的不同：

- 等待队列入口设置了WQ_FLAG_EXCLUSIEV标志时，则会被添加到等待队列的尾部。而没有这个标志的入口会被添加到头部。
- 在某个等待队列上调用*wake_up*时，它会在唤醒第一个具有WQ_FLAG_EXCLUSIEVE标志的进程之后停止唤醒其他进程。

最终结果是，执行独占等待的进程每次只会被唤醒其中一个（以某种有序的方式），从而不会产生“疯狂兽群”问题。但是，内核每次仍然会唤醒所有非独占等待进程。

如果满足下面两个条件，在驱动程序中利用独占等待是值得考虑的。这两个条件是：对某个资源存在严重竞争，并且唤醒单个进程就能完整消耗该资源。比如，对Apache Web服务器来说，独占等待就非常适合；当新的连接到来时，只应有一个Apache进程（通常有许多Apache进程）被唤醒并处理该连接。但是，我们并没有在*scullpipe*驱动程序中使用独占等待，因为读取进程很少会竞争数据（或者写入进程竞争缓冲区空间），另外我们也不知道某个读取进程在唤醒后，是不是会消耗所有的可用数据。

将进程置于可中断等待状态是调用*prepare_to_wait_exclusive*的一种简单方式：

```
void prepare_to_wait_exclusive(wait_queue_head_t *queue,
                               wait_queue_t *wait,
                               int state);
```

这个函数可用来替换*prepare_to_wait*函数，它设置等待队列入口的“独占”标志，并将进程添加到等待队列的尾部。注意，使用*wait_event*及其变种无法执行独占等待。

唤醒的相关细节

相比内核中真正发生的事情，我们介绍过的唤醒进程的内容要简单多了。当一个进程被唤醒时，实际的结果由等待队列入口中的一个函数控制。默认的唤醒函数（注3）将进程设置为可运行状态，并且如果该进程具有更高的优先级，则会执行一次上下文切换以便切换到该进程。设备驱动程序基本没有必要提供不同的唤醒函数。如果读者认为有此必要，可参阅 *<linux/wait.h>* 获得关于如何设置的信息。

我们也没有看到*wake_up*的所有变种。大部分驱动程序开发者从来不需要这些变种，但出于完整性，这里给出所有的 *wake_up* 变种：

```
wake_up(wait_queue_head_t *queue);
wake_up_interruptible(wait_queue_head_t *queue);
```

*wake_up*会唤醒队列上所有非独占等待的进程，以及单个独占等待者（如果存在）。*wake_up_interruptible* 完成相同的工作，只是它会跳过不可中断休眠的那些进程。这两个函数会在返回前让唤醒的一个或者多个进程被调度（尽管这种情况在原子上下文中调用时不会发生）。

```
wake_up_nr(wait_queue_head_t *queue, int nr);
wake_up_interruptible_nr(wait_queue_head_t *queue, int nr);
```

上述函数的功能和*wake_up*类似，只是它们只会唤醒nr个独占等待进程，而不是只有一个。注意，传递0表明请求唤醒所有的独占等待进程，而不是不唤醒任何一个。

```
wake_up_all(wait_queue_head_t *queue);
wake_up_interruptible_all(wait_queue_head_t *queue);
```

上述形式的*wake_up*不管进程是否执行独占等待均唤醒它们（尽管中断形式仍然会跳过执行非中断等待的进程）。

```
wake_up_interruptible_sync(wait_queue_head_t *queue);
```

通常，被唤醒的进程可能会抢占当前的进程，并在*wake_up*返回前被调度到处理器上。换句话说，对*wake_up*的调用可能不是原子的。如果调用*wake_up*的进程运行在原子上下文（例如拥有自旋锁，或者是一个中断处理例程）中，则重新调度就不会发生。通常，这一保护是适当的。但是，如果你不希望在这时被调度出处理器，则可使用*wake_up_interruptible*的“sync（同步）”变种。这一函数经常在调用者打算强制重新调度的情况下使用，并且在只有很少的工作需要首先完成时更加有效。

注3：它有一个虚构的名字*default_wake_function*。

如果读者感觉上面描述的东西不是非常清晰，不要着急。除了*wake_up_interruptible*之外，很少有驱动程序需要调用其他*wake_up*函数。

旧的历史：sleep_on

如果读者花些时间深入到内核源代码，可能会遇到我们尚未讨论过的两个函数：

```
void sleep_on(wait_queue_head_t *queue);
void interruptible_sleep_on(wait_queue_head_t *queue);
```

读者可以能会想到，这两个函数将当前进程无条件休眠在给定的队列上。但是，我们极不赞成使用这两个函数——永远不要使用它们。如果考虑下面的情形，就能清楚知道问题所在：*sleep_on*没有提供对竞态的任何保护方法。在代码决定休眠及*sleep_on*真正产生作用之间，总是存在一个窗口，而窗口期间出现的唤醒将会被丢失。为此，调用*sleep_on*的代码整体上是不安全的。

在不远的将来，对*sleep_on*及其变种（还有两个超时形式未列出）的调用将从内核中删除。

测试 Scullpipe 驱动程序

我们已经看到*sullpipe*驱动程序实现阻塞I/O的方式。如果读者打算试试这个驱动程序，则其源代码可和本书其他示例程序一起找到。实际的阻塞I/O可通过打开两个窗口看到。第一个窗口中运行类似*cat/dev/scullpipe*这样的命令，然后在另一窗口中复制一个文件到*/dev/scullpipe*，之后将看到该文件的内容会出现在第一个窗口中。

测试非阻塞活动要麻烦些，因为一般的程序不会做非阻塞型操作。*misc-progs*源码目录中包含一个简单的程序*nbtest*，用来测试非阻塞型操作，其代码罗列如下。它所做的全部事情就是用非阻塞型I/O把输入复制到输出，并在期间稍做延迟。延迟时间由命令行传递，默认是1秒。

```
int main(int argc, char **argv)
{
    int delay = 1, n, m = 0;

    if (argc > 1)
        delay=atoi(argv[1]);
    fcntl(0, F_SETFL, fcntl(0,F_GETFL) | O_NONBLOCK); /* stdin */
    fcntl(1, F_SETFL, fcntl(1,F_GETFL) | O_NONBLOCK); /* stdout */

    while (1) {
        n = read(0, buffer, 4096);
        if (n >= 0)
           m = write(1, buffer, n);
```

```
            if ((n < 0 || m < 0) && (errno != EAGAIN))
                break;
            sleep(delay);
        }
        perror(n < 0 ? "stdin" : "stdout");
        exit(1);
    }
```

如果读者在诸如*strace*这类进程跟踪工具下运行上面的程序，则会看到每个操作的成功或失败，这取决于操作发生时数据是否可获得。

poll 和 select

使用非阻塞I/O的应用程序也经常使用*poll*、*select*和*epoll*系统调用。*poll*、*select*和*epoll*的功能本质上是一样的：都允许进程决定是否可以对一个或多个打开的文件做非阻塞的读取或写入。这些调用也会阻塞进程，直到给定的文件描述符集合中的任何一个可读取或写入。因此，它们常常用于那些要使用多个输入或输出流而又不会阻塞于其中任何一个流的应用程序中。同一功能之所以要由多个独立的函数提供，是因为其中两个几乎是同时由两个不同的Unix团体分别实现的：*select*在BSD Unix中引入，而*poll*由System V引入。*epool*系统调用（注4）在2.5.45中引入，它用于将*poll*函数扩展到能够处理数千个文件描述符。

对上述系统调用的支持需要来自设备驱动程序的相应支持。所有三个系统调用均通过驱动程序的*poll*方法提供。该方法具有如下的原型：

```
unsigned int (*poll) (struct file *filp, poll_table *wait);
```

当用户空间程序在驱动程序关联的文件描述符上执行*poll*、*select*或*epoll*系统调用时，该驱动程序方法将被调用。该设备方法分为两步处理：

1. 在一个或多个可指示poll状态变化的等待队列上调用*poll_wait*。如果当前没有文件描述符可用来执行I/O，则内核将使进程在传递到该系统调用的所有文件描述符对应的等待队列上等待。
2. 返回一个用来描述操作是否可以立即无阻塞执行的位掩码。

这些操作通常简单明了，各个驱动程序的这些操作看起来也非常类似。然而，实际上它们依赖于只有驱动程序才能提供的信息，因此必须为每个驱动程序分别实现对应的操作。

注4：实际上，*epoll*是用于实现轮询功能的三个调用的集合。对我们来说，可以简单地认为*epoll*就是一个单独的调用。

传递给*poll*方法的第二个参数，poll_table结构，用于在内核中实现*poll*、*select*以及*epool*系统调用。它在<*linux/poll.h*>中声明，驱动程序源代码必须包含这个头文件。驱动程序编写者不需要了解该结构的细节，且需要将其当成一个不透明对象使用。它被传递给驱动程序方法，以使每个可以唤醒进程和修改*poll*操作状态的等待队列都可以被驱动程序装载。通过*poll_wait*函数，驱动程序向poll_table结构添加一个等待队列：

```
void poll_wait (struct file *, wait_queue_head_t *, poll_table *);
```

*poll*方法执行的第二项任务是返回描述哪个操作可以立即执行的位掩码，这也很直接，例如，如果设备已有数据就绪，一个*read*操作可以立刻完成而不用休眠，那么*poll*方法应该指出这种情况。几个标志（在<*linux/poll*.h>定义）用来指明可能的操作：

POLLIN
: 如果设备可以无阻塞地读取，就设置该位。

POLLRDNORM
: 如果“通常”的数据已经就绪，可以读取，就设置该位。一个可读设备返回(POLLIN | POLLRDNORM)。

POLLRDBAND
: 这一位指示可以从设备读取out-of-band(频带之外)的数据。它当前只可以在Linux内核的DECnet代码中使用，通常不用于设备驱动程序。

POLLPRI
: 可以无阻塞地读取高优先级（即out-of-band）的数据。设置该位会导致*select*报告文件发生一个异常，这是由于*select*把“out-of-band”的数据作为异常对待。

POLLHUP
: 当读取设备的进程到达文件尾时，驱动程序必须设置POLLHUP（挂起）位。依照*select*的功能描述，调用*select*的进程会被告知设备是可读的。

POLLERR
: 设备发生了错误。如果调用*poll*，就会报告设备既可读也可以写，因为读写都会无阻塞地返回一个错误码。

POLLOUT
: 如果设备可以无阻塞地写入，就在返回值中设置该位。

POLLWRNORM
: 该位和POLLOUT的意义一样，有时其实就是同一个数字。一个可写的设备将返回(POLLOUT | POLLWRNORM)。

POLLWRBAND

与POLLRDBAND类似，这一位表示具有非零优先级的数据可以被写入设备。只有数据报（datagram）的*poll*实现中使用了这一位，因为数据报可以传输out-of-band数据。

POLLRDBAND和POLLWRBAND只在与套接字相关的文件描述符中才是有意义的。设备驱动程序通常用不到这两个标志。

描述*poll*很费事，实际的使用则相对简单多了。考虑一下*scullpipe*的*poll*实现：

```
static unsigned int scull_p_poll(struct file *filp, poll_table *wait)
{
    struct scull_pipe *dev = filp->private_data;
    unsigned int mask = 0;

    /*
     * The buffer is circular; it is considered full
     * if "wp" is right behind "rp" and empty if the
     * two are equal.
     */
    /*
     * 缓冲区是环形的；也就是说，如果wp在rp之后，则表明缓冲区
     * 已满，而如果它们两个相等，则表明是空的。
     */
    down(&dev->sem);
    poll_wait(filp, &dev->inq,  wait);
    poll_wait(filp, &dev->outq, wait);
    if (dev->rp != dev->wp)
        mask |= POLLIN | POLLRDNORM;    /* 可读取 */
    if (spacefree(dev))
        mask |= POLLOUT | POLLWRNORM;   /* 可写入 */
    up(&dev->sem);
    return mask;
}
```

这段代码简单地增加两个*scullpipe*等待队列到poll_table中，然后根据数据的可读或可写状态设置相应的位掩码。

上述*poll*代码中缺少对文件尾的支持，这是因为*scullpipe*不支持文件尾条件。对大多数真实的设备，在目前（或将来）没有数据可获得时，*poll*方法应该返回POLLHUP。如果调用者使用*select*系统调用，则会报告文件是可读的。在两种情况下应用程序都能知道它一定可以执行无阻塞的*read*，而*read*方法将会返回0来指示已到了文件尾。

在真正的FIFO实现中，读取进程在所有的写入进程都关闭了文件后就能看到文件尾。然而在*scullpipe*中读取进程却永远看不到文件的结尾。之所以有这种不同，是因为FIFO一般被作为两个进程的通信通道使用，而*scullpipe*就是一个垃圾桶——只要还有一个

读取进程存在，任何进程都可以往里面扔数据。此外，重新实现内核中已有的东西也没什么意义，因此，我们在例子中采用了不同的实现行为。

像FIFO那样实现文件尾意味着要在*read*和*poll*中检查dev->nwrites，如果没有进程为写入而打开设备，就报告文件结束。不过遗憾的是，如果读取进程在写入进程之前打开了*scullpipe*设备，马上就会看到文件尾，从而没有机会等待数据的到达。修正这个问题的最好方法是像真正的FIFO那样实现阻塞的*open*操作。这个任务作为练习留给读者。

与 read 和 write 的交互

*poll*和*select*调用的目的是确定接下来的I/O操作是否会阻塞。从这个方面来说，它们是*read*和*write*的补充。*poll*和*select*的更重要的用途是它们可以使应用程序同时等待多个数据流，尽管在*scull*的例子里没有利用这个特点。

为了使应用程序正常工作，正确实现这三个调用是非常重要的。所以尽管下面的规则多多少少已经在前面提过了，但我们还是要在这里总结一下。

从设备读取数据

- 如果输入缓冲区有数据，那么即使就绪的数据比程序所请求的少，并且驱动程序保证剩下的数据马上就能到达，*read*调用仍然应该以难以察觉的延迟立即返回。如果为了某种方便(比如我们的*scull*)，*read*甚至可以一直返回比所请求数目少的数据，当然，前提是至少得返回一个字节。
- 如果输入缓冲区中没有数据，那么默认情况下*read*必须阻塞等待，直到至少有一个字节到达。另一方面，如果设置了O_NONBLOCK标志，*read*应立即返回，返回值是-EAGAIN（有些System V的老版本返回0）。在这种情况下*poll*必须报告设备不可读，直到至少有一个字节到达。一旦缓冲区中有了数据，我们就回到了前一种情况。
- 如果已经到达文件尾，*read*应该立即返回0，无论O_NONBLOCK是否设置。此时*poll*应该报告POLLHUP。

向设备写数据

- 如果输出缓冲区中有空间，则*write*应该无延迟地立即返回。它可以接收比请求少的数据，但至少要接收一个字节。在这种情况下，*poll*报告设备可写。
- 如果输出缓冲区已满，那么默认情况下*write*被阻塞直到有空间释放。如果设置了O_NONBLOCK标志，*write*应立即返回，返回值是-EAGAIN（老版本的System V

系统返回0)。这时*poll*应该报告文件不可写。另一方面，如果设备不能再接受任何数据，则*write*返回-ENOSPC（"No space left on device，设备无可用空间"），而不管O_NONBLOCK标志是否设置。

- 永远不要让*write*调用在返回前等待数据的传输结束，即使O_NONBLOCK标志被清除。这是因为，许多应用程序用*select*来检查*write*是否会阻塞。如果报告设备可以写入，调用就不能被阻塞。如果使用设备的程序需要保证输出缓冲区中的数据确实已经被传送出去，驱动程序就必须提供一个*fsync*方法。例如，可移除设备就应该有一个*fsync*的入口点。

尽管这些已经是一个很好的通用规则集合，但还是应该承认每个设备都有其独特之处，所以有时候需要稍稍改变一下规则。例如，面向记录的设备（如磁带机）不能执行部分写入（必须以记录为单位）。

刷新待处理输出

我们已经看到了为什么*write*方法不能满足所有数据输出的需求，*fsync*函数可以弥补这一空隙，它通过同名系统调用来调用。该方法的原型是：

```
int (*fsync) (struct file *file, struct dentry *dentry, int datasync);
```

如果应用程序需要确保数据已经被传送到设备上，就必须实现*fsync*方法。一个*fsync*调用只有在设备已被完全刷新（输出缓冲区全空）时才会返回，即使这要花一些时间。是否设置了O_NONBLOCK标志对此没有影响。参数datasync用于区分*fsync*和*fdatasync*这两个系统调用。这里它只和文件系统的代码有关，驱动程序可以忽略它。

*fsync*方法没有什么特别的地方。这个调用对时间没有严格要求，所以每个驱动程序都可以按照作者的喜好实现它。大多数时候，字符设备驱动程序在它们的fops只有一个NULL指针，而块设备总是用通用的*block_fsync*来实现这个方法，*block_fsync*会依次刷新设备的所有缓冲块，并等待所有I/O结束。

底层的数据结构

*poll*和*select*系统调用的实现是相当简单的；*epoll*略微复杂些，但仍基于相同的原理。当用户应用程序调用了*poll*、*select*或*epoll_ctl*（注5）函数时，内核会调用由该系统调用引用的全部文件的*poll*方法，并向它们传递同一个poll_table。poll_table结构是构成实际数据结构的一个简单封装。对*poll*和*select*系统调用，后面这个结构是包含

注5：这是为调用*epoll_wait*而用于构造内部数据结构的函数。

poll_table_entry结构的内存页链表。每个poll_table_entry结构包括一个指向被打开设备的 struct file 类型的指针、一个 wait_queue_head_t 指针以及一个关联的等待队列入口。对*poll_wait*的调用有时也会将进程添加到这个给定的等待队列。整个结构必须由内核维护，因而在*poll*或*select*返回前，进程可从所有这些队列中移除。

如果轮询（poll）时没有一个驱动程序指明可以进行非阻塞I/O，这个*poll*调用就进入休眠，直到休眠在其上的某个（或多个）等待队列唤醒它为止。

*poll*实现中的一个有趣之处是，驱动程序的*poll*方法在被调用时为poll_table参数传递 NULL 指针。这种情形的出现基于两个原因。如果应用程序调用 *poll* 时提供的超时（timeout）值为 0（表明不应等待），则没有任何原因需要处理等待队列，而系统也不必为此做任何工作。在任何被轮询的驱动程序指明可进行 I/O 之后，poll_table 指针也会立即被设置为 NULL。因为内核知道此时不会发生任何等待，因此也不需要构造等待队列。

在*poll*调用结束时，poll_table结构被重新分配，所有的先前添加到 poll 表中的等待队列入口都会从这个表以及等待队列中移除。

在图 6-1 中显示了与轮询相关的数据结构；该图是实际数据结构的简化表示，其中忽略了轮询表的多页特性，也省略了每个poll_table_entry中的文件指针。推荐对真正的实现感兴趣的读者阅读 *<linux/poll.h>* 和 *fs/select.c* 的相关代码。

此时，读者应该能够理解出现新的 *epoll* 系统调用的原因了。在典型情况下，对 *poll* 或者 *select* 的调用只涉及到几个文件描述符，因此构造相关数据结构的成本相对较小。但是有一些应用程序却会同时处理几千个文件描述符。对这种情况，在每个 I/O 操作之间构造及销毁该数据结构将变得极其浪费。而 *epoll* 系统调用族可帮助这类应用程序只需构造一次内部内核数据结构，然后多次使用。

异步通知

尽管大多数时候阻塞型和非阻塞型操作的组合以及 *select* 方法可以有效地查询设备，但某些时候用这种技术处理就效率不高了。

例如，我们可以想像这种情况：一个进程在低优先级执行长的循环计算，但又需要尽可能快地处理输入数据。如果该进程正在响应来自数据收集外设的新的观测数据，则应该在新数据可用时立即知晓并处理。我们可以让这个应用程序周期性地调用*poll*来检查数据，但是对许多情况来讲还有更好的办法。通过使用异步通知，应用程序可以在数据可用时收到一个信号，而不需要不停地使用轮询来关注数据。

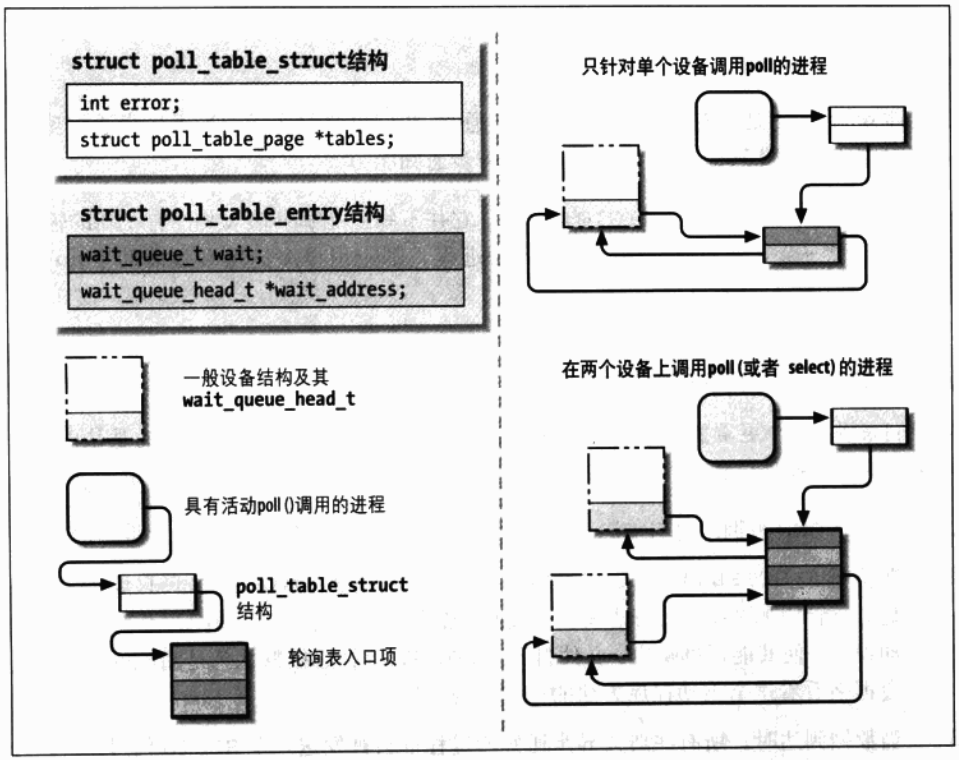


图 6-1：poll 背后的数据结构

为了启用文件的异步通知机制，用户程序必须执行两个步骤。首先，它们指定一个进程作为文件的“属主（owner）”。当进程使用*fcntl*系统调用执行F_SETOWN命令时，属主进程的进程ID号就被保存在filp->f_owner中。这一步是必需的，目的是为了让内核知道应该通知哪个进程。然后，为了真正启用异步通知机制，用户程序还必须在设备中设置FASYNC标志，这通过*fcntl*的F_SETFL命令完成的。

执行完这两个步骤之后，输入文件就可以在新数据到达时请求发送一个SIGIO信号。该信号被发送到存放在filp->f_owner中的进程（如果是负值就是进程组）。

例如，用户程序中的如下代码段启用了stdin输入文件到当前进程的异步通知机制：

```
signal(SIGIO, &input_handler); /* 虚构示例，最好使用 sigaction() */
fcntl(STDIN_FILENO, F_SETOWN, getpid());
oflags = fcntl(STDIN_FILENO, F_GETFL);
fcntl(STDIN_FILENO, F_SETFL, oflags | FASYNC);
```

示例源代码中名为*asynctest*的程序就是这样一个读取stdin的例子。它可以用来测试

scullpipe 的异步功能。该程序类似于 *cat*，但不会在文件尾终止。它只响应输入，没有输入时就没有响应。

需要注意的是，不是所有的设备都支持异步通知，我们也可以选择不提供异步通知功能。应用程序通常假设只有套接字和终端才有异步通知能力。

还有一个问题。当进程收到 SIGIO 信号时，它并不知道是哪个输入文件有了新的输入。如果有多于一个文件可以异步通知输入的进程，则应用程序仍然必须借助于 *poll* 或 *select* 来确定输入的来源。

从驱动程序的角度考虑

对我们来讲，一个更重要的话题是驱动程序怎样实现异步信号。下面列出的是从内核角度来看的详细操作过程。

1. F_SETOWN 被调用时对 filp->f_owner 赋值，此外什么也不做。
2. 在执行 F_SETFL 启用 FASYNC 时，调用驱动程序的 *fasync* 方法。只要 filp->f_flags 中的 FASYNC 标志发生了变化，就会调用该方法，以便把这个变化通知驱动程序，使其能正确响应。文件打开时，FASYNC 标志被默认为是清除的。我们一会再来看看这个驱动程序方法的标准实现。
3. 当数据到达时，所有注册为异步通知的进程都会被发送一个 SIGIO 信号。

第一步的实现很简单，在驱动程序部分没什么可做的。其他步骤则要涉及维护一个动态数据结构，以跟踪不同的异步读取进程，这种进程可能会有好几个。不过，这个动态数据结构并不依赖于特定的设备，内核已经提供了一套合适的通用实现方法，无需为每个驱动程序重写同一代码。

Linux 的这种通用方法基于一个数据结构和两个函数（它们要在前面提到的第二步和第三步中调用）。含有相关声明的头文件是 *<linux/fs.h>*（这对我们来说并不新鲜），那个数据结构称为 struct fasync_struct。和处理等待队列的方式类似，我们需要把一个该类型的指针插入设备特定的数据结构中去。

驱动程序要调用的两个函数的原型如下：

```
int fasync_helper(int fd, struct file *filp,
        int mode, struct fasync_struct **fa);
void kill_fasync(struct fasync_struct **fa, int sig, int band);
```

当一个打开的文件的FASYNC标志被修改时，调用fasync_helper以便从相关的进程列表中增加或删除文件。除了最后一个参数外，它的其他所有参数都是提供给*fasync*方法的相同参数，因此可以直接传递。在数据到达时，可使用kill_fasync通知所有的相关进程。它的参数包括要发送的信号（通常是SIGIO）和带宽（band），后者几乎总是POLL_IN（注6）（但在网络代码中，可用来发送“紧急”或out-of-band的数据）。

在*scullpipe*中是这样实现*fasync*方法的：

```
static int scull_p_fasync(int fd, struct file *filp, int mode)
{
    struct scull_pipe *dev = filp->private_data;

    return fasync_helper(fd, filp, mode, &dev->async_queue);
}
```

很显然，所有工作都由*fasync_helper*完成。不过，如果没有驱动程序中提供的方法，它是不可能实现这一功能的。因为辅助函数需要访问正确的struct fasync_struct *类型（这里是&dev->async_queue）的指针，而只有驱动程序才能提供这一信息。

接着，当数据到达时，必须执行下面的语句来通知异步读取进程。由于供给*scullpipe*的读取进程的新数据是由某个进程调用*write*产生的，所以这条语句是在*scullpipe*的*write*方法中：

```
if (dev->async_queue)
    kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
```

注意，某些设备也针对设备可写入而实现了异步通知。在这种情况下，*kill_fasync*必须以POLL_OUT为模式调用。

看起来差不多都讨论完了，不过还漏了一件事。当文件关闭时必须调用*fasync*方法，以便从活动的异步读取进程列表中删除该文件。尽管这个调用只在filp->f_flags设置了FASYNC标志时才是必需的，但不管什么情况，调用它不会有什么坏处，并且这也是普遍的实现方法。例如，下面的代码是*scullpipe*的*close*方法中的一段：

```
/* 从异步通知列表中删除该filp */
scull_p_fasync(-1, filp, 0);
```

异步通知所使用的数据结构和struct wait_queue使用的几乎是相同的，因为两种情况都涉及等待事件。不同之处在于前者用struct file替换了struct task_struct。队列中的file结构用来获取f_owner，以便给进程发送信号。

注6：POLL_ZN是异步通知代码使用的一个符号，它等价于POLLIN|POLLRDNORM。

定位设备

本章最后要讨论的是 *llseek* 方法、对某些设备来讲，该方法很有用而且易于实现。

llseek 实现

llseek 方法实现了 *lseek* 和 *llseek* 系统调用。前面已经提到过，如果设备操作未定义 *llseek* 方法，内核默认通过修改 `filp->f_pos` 而执行定位，`filp->f_pos` 是文件的当前读取/写入位置。请注意，为了使 *lseek* 系统调用能正确工作，*read* 和 *write* 方法必须通过更新它们收到的偏移量参数来配合。

如果定位操作对应于设备的一个物理操作，可能就需要提供自己的 *llseek* 方法。在 *scull* 的驱动程序中可以看到一个简单的例子：

```
loff_t scull_llseek(struct file *filp, loff_t off, int whence)
{
    struct scull_dev *dev = filp->private_data;
    loff_t newpos;

    switch(whence) {
      case 0: /* SEEK_SET */
        newpos = off;
        break;

      case 1: /* SEEK_CUR */
        newpos = filp->f_pos + off;
        break;

      case 2: /* SEEK_END */
        newpos = dev->size + off;
        break;

      default: /* 不应该发生 */
        return -EINVAL;
    }
    if (newpos < 0) return -EINVAL;
    filp->f_pos = newpos;
    return newpos;
}
```

这里唯一与设备相关的操作就是从设备中获得文件长度。在 *scull* 中，*read* 和 *write* 方法需要相互配合，就像第三章中介绍的那样。

上面的实现对 *scull* 是有意义的，因为它处理一个明确定义的数据区。然而大多数设备只提供了数据流（就像串口和键盘），而不是数据区，定位这些设备是没有意义的。在这种情况下，不能简单地不声明 *llseek* 操作，因为默认方法是允许定位的。相反，我们应该在我们的 *open* 方法中调用 *nonseekable_open*，以便通知内核设备不支持 *llseek*：

```
int nonseekable_open(struct inode *inode; struct file *filp);
```

上述调用将会把给定的*filp*标记为不可定位；这样，内核就不会让这种文件上的*lseek*调用成功。通过这种方式标记文件，我们还可以确保通过*pread*和*pwrite*系统调用也不能定位文件。

为了完整起见，我们还应该将file_operations结构中的*llseek*方法设置为特殊的辅助函数*no_llseek*，该函数定义在*<linux/fs.h>*中。

设备文件的访问控制

提供访问控制对于设备节点的可靠性有时是至关重要的。比如，不仅不允许未授权的用户使用设备（这可以通过设置文件系统的许可位实现），而且在某些情况下一次只能允许一个授权用户打开设备。

使用终端的问题与此类似。每当一个用户登录系统，*login*进程就修改设备节点的属主，以防止其他用户干扰或侦听这个终端的数据流。然而如果仅仅为了保证对设备的唯一访问，而在每次打开它时都用特权程序修改设备的属主，是不现实的。

到现在为止，我们还没有看到能超越文件系统权限位而实现任意访问控制的代码。如果*open*系统调用将请求转给驱动程序，*open*就成功了。现在来介绍一些实现某些附加检查的技术。

本节的每个设备都和“裸的”scull设备（它实现了一个持久的内存区）具有相同的功能，但具有不同的访问控制，这是在*open*和*release*操作中实现的。

独享设备

最生硬的访问控制方法是一次只允许一个进程打开设备（独享）。最好避免使用这种技术，因为它制约了用户的灵活性。用户可能会希望在同一设备上运行不同的进程，一个用来读取状态信息，而另一个进程写入数据。有时候，用户通过一个shell脚本同时运行多个可以同时访问设备的简单程序就能够完成很多工作。换句话说，独享这种方式其实建立了一种策略，而这种策略妨碍了用户完成他的工作。

一次只允许一个进程打开设备有很多令人不快的特性，不过这也是设备驱动程序中最容易实现的访问控制方法。下面给出了代码，这些代码摘自*scullsingle*设备。

*scullsingle*设备维护一个atomic_t变量，称为scull_s_available。该变量的值初始化为1，表明该设备真正可用。*open*调用会减小并测试scull_s_availabe，并在其他进程已经打开该设备时拒绝访问：

```
static atomic_t scull_s_available = ATOMIC_INIT(1);

static int scull_s_open(struct inode *inode, struct file *filp)
{
    struct scull_dev *dev = &scull_s_device; /* 设备信息 */

    if (! atomic_dec_and_test (&scull_s_available)) {
        atomic_inc(&scull_s_available);
        return -EBUSY; /* 已打开 */
    }

    /* 然后，从裸的scull设备中复制所有其他数据 */
    if ( (filp->f_flags & O_ACCMODE) =   = O_WRONLY)
        scull_trim(dev);
    filp->private_data = dev;
    return 0;           /* 成功 */
}
```

另一方面，*release* 调用则标记设备为不再忙。

```
static int scull_s_release(struct inode *inode, struct file *filp)
{
    atomic_inc(&scull_s_available); /* 释放该设备 */
    return 0;
}
```

通常，建议把打开标志scull_s_available放在设备结构（这里是Scull_Dev）中，因为从概念上来讲它本身属于设备。不过，*scull*驱动程序使用了单独的变量保存这个标志，这是为了保持与裸*scull*设备使用同样的设备结构和方法，从而最小化代码的重复。

限制每次只由一个用户访问

在构造独享设备之后，我们要建立允许单个用户在多个进程中打开的设备，但是每次只允许一个用户打开该设备。这种方案便于测试该设备，因为用户每次可从多个进程读取和写入，前提是由用户负责在多进程访问中维护数据的完整性。这通过在*open*方法中加入检查来完成，这种检查在正常的权限检查之后进行，提供了比文件属主和属组权限位更严格的访问控制。这种策略和终端使用的访问策略相同，不过它无需借助于一个外部的特权程序。

与独享策略相比，实现这些访问策略需要更多的技巧。此时需要两个数据项：一个打开计数和设备属主的UID。同样的，这些数据项最好是保存在设备结构内部；不过，我们的例子使用的是全局变量，其原因在前面介绍*scullsingle*时已解释过了。设备名是*sculluid*。

*open*调用在第一次打开时授权，但它记录下设备的属主。这意味着一个用户可以多次打开设备，允许几个互相协作的进程并发地在设备上操作。同时，其他用户不能打开这个

设备，这就避免了外部干扰。因为这个函数版本和上一个基本相同，所以只列出相关部分：

```
spin_lock(&scull_u_lock);
if (scull_u_count &&
        (scull_u_owner != current->uid) &&   /* 允许用户 */
        (scull_u_owner != current->euid) && /* 允许执行 su 命令的用户 */
        !capable(CAP_DAC_OVERRIDE)) { /* 也允许 root 用户 */
    spin_unlock(&scull_u_lock);
    return -EBUSY;    /* 返回 -EPERM 会让用户混淆 */
}

if (scull_u_count ==  = 0)
    scull_u_owner = current->uid; /* 获得所有者 */

scull_u_count++;
spin_unlock(&scull_u_lock);
```

注意，*sculluid* 代码有两个变量（scull_u_owner 和 scull_u_count），这两个变量控制对设备的访问，并且可由多个进程并发地访问。为了让这些变量安全，我们通过一个自旋锁（scull_u_lock）来保护对这些变量的访问。没有这个锁，两个（或更多）的进程可能在同一时刻测试scull_u_count，而它们均可能做出可获得设备所有权的结论。这里采用自旋锁的原因在于，锁的拥有时间将非常短，而在拥有锁的时间内，驱动程序不会做任何可能休眠的工作。

即使代码执行的是权限检查，但我们还是选择返回 -EBUSY 而不是 -EPERM，以便给被拒绝访问的用户提示正确的信息。返回“权限拒绝（Permission denied）”通常是检查 */dev* 文件的模式和属主的结果，而“设备忙（Device busy）”提示用户设备已经被进程使用。

代码还检查了试图打开设备的进程是否有越过文件访问权限的能力；如果是这样，允许它进行打开操作，即使这个进程不是设备属主。在这种情况下，CAP_DAC_OVERRIDE 正适合于完成这项任务。

release 方法如下实现：

```
static int scull_u_release(struct inode *inode, struct file *filp)
{
    spin_lock(&scull_u_lock);
    scull_u_count--; /* 除此之外不做任何事情 */
    spin_unlock(&scull_u_lock);
    return 0;
}
```

我们再次看到，在修改计数之前，我们必须获取自旋锁，这样就不会和其他进程发生竞争。

替代 EBUSY 的阻塞型 open

当设备不能访问时返回一个错误，通常这是最合理的方式，但有些情况下可能需要让进程等待设备。

例如，如果一个以周期性的、预定的方式发送定时报告的数据通道，同时也能根据人们的需要而临时使用，那么在通道正忙的时候，定时报告最好能够稍微延迟一会儿，而不是因为通道忙就返回失败。

这是在设计设备驱动程序时程序员必须作出的选择，所解决的问题不同，答案也就不一样。

读者可能已经想到，代替 EBUSY 的另一个方法是实现阻塞型 *open*。*scullwuid* 设备和 *sculluid* 的不同是，*open* 时会等待设备而不是返回 -EBUSY。它和 *sculluid* 只在 *open* 操作的下列部分不同：

```
spin_lock(&scull_w_lock);
while (! scull_w_available()) {
    spin_unlock(&scull_w_lock);
    if (filp->f_flags & O_NONBLOCK) return -EAGAIN;
    if (wait_event_interruptible (scull_w_wait, scull_w_available()))
        return -ERESTARTSYS; /* 告诉fs层做进一步处理 */
    spin_lock(&scull_w_lock);
}
if (scull_w_count =  = 0)
    scull_w_owner = current->uid; /* 获取所有者 */
scull_w_count++;
spin_unlock(&scull_w_lock);
```

这里的实现又是基于等待队列。创建等待队列是为了维护一个因等待事件而休眠的进程的列表，所以在这里使用非常合适。

接下来，*release* 方法唤醒所有等待的进程：

```
static int scull_w_release(struct inode *inode, struct file *filp)
{
    int temp;

    spin_lock(&scull_w_lock);
    scull_w_count--;
    temp = scull_w_count;
    spin_unlock(&scull_w_lock);
    if (temp =  = 0)
        wake_up_interruptible_sync(&scull_w_wait); /* 唤醒其他的uid进程 */
    return 0;
}
```

上面恰好给出了调用*wake_up_interruptible_sync*函数的一个例子。在执行这个唤醒时，我们正打算返回到用户空间，而这恰好就是系统的调度点。因此，在执行这个唤醒时，我们没有必要引发潜在的重新调度，而只需要调用“sync”版本并结束我们的工作。

阻塞型*open*实现中的问题是，对于交互式用户来说它是令人很不愉快的，用户可能会在等待中猜测设备出了什么问题。交互式用户通常使用诸如*cp*和*tar*这样的预先编译好的命令，它们都没有在*open*调用中加入O_NONBLOCK选项。隔壁正使用磁带机做备份的同事可能更愿意得到一条清晰的消息“设备或资源忙”，而不是在*tar*命令扫描磁盘的时候坐在一边猜想为什么今天的硬盘这么安静。

这类问题（对同一设备的不同的、不兼容的策略）最好通过为每一种访问策略实现一个设备节点的方法来解决。这种实现的一个例子是Linux的磁带设备驱动程序，它为同一个设备提供了多个设备文件。不同的设备文件会使设备以不同的方式工作，例如是否以压缩方式记录、在设备关闭时是否自动回卷磁带，等等。

在打开时复制设备

另一个实现访问控制的方法是，在进程打开设备时创建设备的不同私有副本。

显然这种方法只有在设备没有绑定到某个硬件对象时才能实现。*scull*就是这样一个“软设备”的例子。*/dev/tty*内部也使用了类似的技术，以提供给它的进程一个不同于*/dev*入口点所表现出的“情景”。如果复制的设备是由软件驱动程序创建的，我们称它们为“虚拟设备”——就像所有的虚拟终端都使用同一个物理终端设备一样。

虽然这种访问控制并不常见，但它的实现展示了内核代码可以轻松地改变应用程序看到的外部环境（如计算机）。

*scull*包中的*/dev/scullpriv*设备节点实现了虚拟设备。在*sucllpriv*的实现中，使用当前进程控制终端的次设备号作为访问虚拟设备的键值。不过这个来源可以很容易地修改成用任意整数值作为键值，不同的键值将导致不同的策略。例如，使用`uid`会导致给每个用户复制不同的虚拟设备，使用`pid`则会导致为每个访问该设备的进程复制一个新设备。

使用控制终端意味着可以通过输入/输出重定向来简化测试设备：运行在某一个虚拟终端的所有命令共享设备，这个设备与在另一个终端上运行的命令所看到的设备互相独立。

*open*方法的代码如下。它必须找到正确的虚拟终端，也许还需要新创建一个。函数的最后一部分没有列出，因为它是从裸*scull*中复制过来的，而这些代码我们已经看到过了。

```
/* 和复制相关的数据结构包括一个 key 成员 */

struct scull_listitem {
    struct scull_dev device;
    dev_t key;
    struct list_head list;
};

/* 设备的链表，以及保护它的锁 */
static LIST_HEAD(scull_c_list);
static spinlock_t scull_c_lock = SPIN_LOCK_UNLOCKED;

/* 查找设备，如果没有就创建一个 */
static struct scull_dev *scull_c_lookfor_device(dev_t key)
{
    struct scull_listitem *lptr;

    list_for_each_entry(lptr, &scull_c_list, list) {
        if (lptr->key =  = key)
            return &(lptr->device);
    }

    /* 没有找到 */
    lptr = kmalloc(sizeof(struct scull_listitem), GFP_KERNEL);
    if (!lptr)
        return NULL;

    /* 初始化该设备 */
    memset(lptr, 0, sizeof(struct scull_listitem));
    lptr->key = key;
    scull_trim(&(lptr->device)); /* 初始化 */
    init_MUTEX(&(lptr->device.sem));

    /* 将其放到链表中 */
    list_add(&lptr->list, &scull_c_list);

    return &(lptr->device);
}

static int scull_c_open(struct inode *inode, struct file *filp)
{
    struct scull_dev *dev;
    dev_t key;

    if (!current->signal->tty) {
        PDEBUG("Process \"%s\" has no ctl tty\n", current->comm);
        return -EINVAL;
    }
    key = tty_devnum(current->signal->tty);

    /* 在链表中查找 scullc 设备 */
    spin_lock(&scull_c_lock);
    dev = scull_c_lookfor_device(key);
    spin_unlock(&scull_c_lock);
```

```
        if (!dev)
            return -ENOMEM;

        /* 然后，从裸的scull设备中复制所有其他数据 */
```

*release*方法没有做什么特殊处理。它通常在最后一次关闭时释放设备，但是为了简化测试，这里没有维护一个打开设备的计数器。如果设备在最后一次关闭时被释放了，则在写入设备后将不能再从中读出同样的数据，除非有一个后台进程保持打开它。我们的示例驱动程序使用了比较简单的方法来保存数据，所以在下一次打开设备时还能找到那些数据。设备在*scull_cleanup*被调用时释放。

上述代码使用了Linux的通用链表机制，而不是自行编写完成相同功能的代码。Linux链表在第十一章中讨论。

这里是*/dev/scullpriv*的*release*的实现。对于设备方法的讨论也到此结束。

```
static int scull_c_release(struct inode *inode, struct file *filp)
{
    /*
     * 因为设备是持久的，所以不需要做任何工作。
     * 一个“真正”的克隆设备应该在最后一次关闭时被释放
     */
    return 0;
}
```

快速参考

本章介绍了下面这些符号和头文件：

```
#include <linux/ioctl.h>
```

这个头文件声明了用于定义*ioctl*命令的所有的宏。它现在包含在*<linux/fs.h>*中。

```
_IOC_NRBITS
_IOC_TYPEBITS
_IOC_SIZEBITS
_IOC_DIRBITS
```

*ioctl*命令的不同位字段的可用位数。还有四个宏定义了不同的MASK（掩码），另外四个宏定义了不同的SHIFT（偏移），但它们基本上仅在内部使用。由于_IOC_SIZEBITS在不同体系架构上的值不同，因此需要重点关注。

_IOC_NONE

_IOC_READ

_IOC_WRITE

"方向"位字段的可能值。"读"和"写"是不同的位，可以"OR"在一起来指定读/写。这些值都是基于0的。

_IOC(dir,type,nr,size)

_IO(type,nr)

_IOR(type,nr,size)

_IOW(type,nr,size)

_IOWR(type,nr,size)

用于生成*ioctl*命令的宏。

_IOC_DIR(nr)

_IOC_TYPE(nr)

_IOC_NR(nr)

_IOC_SIZE(nr)

用于解码*ioctl*命令的宏。特别地，_IOC_TYPE(nr)是_IOC_READ和_IOC_WRITE进行"OR"的结果。

```
#include <asm/uaccess.h>
int access_ok(int type, const void *addr, unsigned long size);
```

这个函数验证指向用户空间的指针是否可用。如果允许访问，*access_ok*返回非零值。

VERIFY_READ

VERIFY_WRITE

在*access_ok*中type参数可取的值。VERIFY_WRITE是VERIFY_READ的超集。

```
#include <asm/uaccess.h>
int put_user(datum,ptr);
int get_user(local,ptr);
int __put_user(datum,ptr);
int __get_user(local,ptr);
```

用于向（或从）用户空间保存（或获取）单个数据项的宏。传送的字节数目由sizeof(*ptr)决定。前两个要先调用*access_ok*，后两个（*__put_user*和*__get_user*）则假设*access_ok*已经被调用过了。

```
#include <linux/capability.h>
```

定义有各种CAP_符号，用于描述用户空间进程可能拥有的权能操作。

```
int capable(int capability);
```

如果进程具有指定的权能，返回非零值。

```
#include <linux/wait.h>
typedef struct { /* ... */ } wait_queue_head_t;
void init_waitqueue_head(wait_queue_head_t *queue);
DECLARE_WAIT_QUEUE_HEAD(queue);
```

预先定义的Linux等待队列类型。wait_queue_head_t类型必须显式地初始化，初始化方法可在运行时用*init_waitqueue_head*，或在编译时用*DECLARE_WAIT_QUEUE_HEAD*。

```
void wait_event(wait_queue_head_t q, int condition);
int wait_event_interruptible(wait_queue_head_t q, int condition);
int wait_event_timeout(wait_queue_head_t q, int condition, int time);
int wait_event_interruptible_timeout(wait_queue_head_t q, int condition,
  int time);
```

使进程在指定的队列上休眠，直到给定的condition值为真。

```
void wake_up(struct wait_queue **q);
void wake_up_interruptible(struct wait_queue **q);
void wake_up_nr(struct wait_queue **q, int nr);
void wake_up_interruptible_nr(struct wait_queue **q, int nr);
void wake_up_all(struct wait_queue **q);
void wake_up_interruptible_all(struct wait_queue **q);
void wake_up_interruptible_sync(struct wait_queue **q);
```

这些函数唤醒休眠在队列q上的进程。*_interruptible*形式的函数只能唤醒可中断的进程。通常，只会唤醒一个独占等待进程，但其行为可通过*_nr*或*_all*形式改变。*_sync*版本的唤醒函数在返回前不会重新调度CPU。

```
#include <linux/sched.h>
set_current_state(int state);
```

设置当前进程的执行状态。TASK_RUNNING表示准备运行，而休眠状态是TASK_INTERRUPTIBLE和TASK_UNINTERRUPTIBLE。

```
void schedule(void);
```

从运行队列中选择一个可运行进程。选定的进程可以是current或另一个不同的进程。

```
typedef struct { /* ... */ } wait_queue_t;
init_waitqueue_entry(wait_queue_t *entry, struct task_struct *task);
```

wait_queue_t类型用来将某个进程放置到一个等待队列上。

```
void prepare_to_wait(wait_queue_head_t *queue, wait_queue_t *wait, int state);
void prepare_to_wait_exclusive(wait_queue_head_t *queue, wait_queue_t *wait,
  int state);
void finish_wait(wait_queue_head_t *queue, wait_queue_t *wait);
```

可用于手工休眠代码的辅助函数。

```
void sleep_on(wiat_queue_head_t *queue);
void interruptible_sleep_on(wiat_queue_head_t *queue);
```

已废弃的两个函数，它们将当前进程无条件地置于休眠状态。

```
#include <linux/poll.h>
void poll_wait(struct file *filp, wait_queue_head_t *q, poll_table *p)
```

将当前进程置于某个等待队列但并不立即调度。该函数主要用于设备驱动程序的*poll*方法。

```
int fasync_helper(struct inode *inode, struct file *filp, int mode, struct
  fasync_struct **fa);
```

用来实现*fasync*设备方法的辅助函数。mode参数取传入该方法的同一值，而fa指向设备专有的fasync_struct *。

```
void kill_fasync(struct fasync_struct *fa, int sig, int band);
```

如果驱动程序支持异步通知，则这个函数可以用来发送一个信号给注册在fa中的进程。

```
int nonseekable_open(struct inode *inode, struct file *filp);
loff_t no_llseek(struct file *file, loff_t offset, int whence);
```

任何不支持定位的设备都应该在其*open*方法中调用*nonseekable_open*。这类设备还应该在其*llseek*方法中使用*no_llseek*。



第七章

时间、延迟及延缓操作

至此，我们已基本知道如何编写一个功能完整的字符模块了。现实中的设备驱动程序除了实现必需的操作外还要做更多工作，如定时、内存管理，硬件访问等等。幸好，内核中提供的许多机制可以简化驱动程序开发者的工作。我们将在后面几章陆续讨论驱动程序可以访问的一些内核资源。在本章中，我们先来看看内核代码是如何对时间问题进行处理的，并按由简到难的顺序逐步讨论，其中包括：

- 如何度量时间差，如何比较时间
- 如何获得当前时间
- 如何将操作延迟指定的一段时间
- 如何调度异步函数到指定的时间之后执行

度量时间差

内核通过定时器中断来跟踪时间流。本书第十章将详细讲述中断的处理。

时钟中断由系统定时硬件以周期性的间隔产生，这个间隔由内核根据HZ的值设定，HZ是一个与体系结构有关的常数，定义在*<linux/param.h>*或者该文件包含的某个子平台相关的文件中。对真实硬件，已发布的Linux内核源代码为大多数平台定义的默认HZ值范围为50～1200，而对软件仿真器的HZ值是24。大多数平台每秒有100或1000次时钟中断，而在常见的x86 PC平台上，默认定义为1000（在包括2.4在内的早期版本中，该平台上的HZ值定义为100）。作为一般性的规则，即使知道对应平台上的确切HZ值，也不应在编程时依赖该HZ值。

如果想改变系统时钟中断发生的频率，可以通过修改HZ值来进行。但是，如果修改了头文件中的HZ值，则必须使用新的值重新编译内核以及所有模块。读者也许想提高HZ

值以在异步任务中获得更细的分辨率，但同时应考虑由此引入的额外时钟开销。实际上，在使用2.4和2.6版本内核的x86工业系统中，将HZ提高到1000是非常常见的。但对当前版本来说，我们应充分信任内核开发者；他们已经为我们选择了最适合的HZ值，因此我们应该保留默认的HZ值而不要自行调整。另外，某些内部计算的实现仅仅适用于HZ取12～1535之间值的情况（见*<linux/timex.h>*和RFC-1589）。

每次当时钟中断发生时，内核内部计数器的值就增加一。这个计数器的值在系统引导时被初始化为0，因此，它的值就是自上次操作系统引导以来的时钟滴答数。这个计数器是一个64位的变量（即使在32位架构上也是64位），称为“jiffies_64”。但是，驱动程序开发者通常访问的是jiffies变量，它是unsigned long型的变量，要么和jiffies_64相同，要么仅仅是jiffies_64的低32位。通常首选使用jiffies，因为对它的访问很快，从而对64位jiffies_64值的访问并不需要在所有架构上都是原子的。

除了由内核管理的低分辨率jiffy机制，某些CPU平台还包含有一个软件可读取的高分辨率计数器。尽管这个计数器的具体使用方法在不同的平台上有所不同，但某些情况下仍是一个非常强大的工具。

使用jiffies计数器

该计数器和读取计数器的工具函数包含在*<linux/jiffies.h>*中，但是通常只需包含*<linux/sched.h>*文件，后者会自动放入*jiffies.h*。还需要说明的是，jiffies和jiffies_64均应被看成只读变量。

在代码需要记录jiffies的当前值时，可简单访问上面说过的unsigned long变量。该变量被声明为volatile，这样可避免编译器对访问该变量的语句的优化。在代码需要计算未来的时间戳时，必须读取当前的计数器，如下例所示：

```
#include <linux/jiffies.h>
unsigned long j, stamp_1, stamp_half, stamp_n;
j = jiffies;                          /* 读取当前值 */
stamp_1    = j + HZ;                  /* 未来的 1 秒 */
stamp_half = j + HZ/2;                /* 半秒 */
stamp_n    = j + n * HZ / 1000;       /* n 毫秒 */
```

只要采用正确的方法来比较不同的值，上述代码就不会因为jiffies的溢出而出现问题。虽然在32位平台上，当HZ取值1000时，每过大约50天该计数器才会溢出一次，但代码仍然应仔细处理这一问题。比较缓存值（比如上面的stamp_1）和当前值时，应该使用下面的宏：

```
#include <linux/jiffies.h>
int time_after(unsigned long a, unsigned long b);
int time_before(unsigned long a, unsigned long b);
int time_after_eq(unsigned long a, unsigned long b);
int time_before_eq(unsigned long a, unsigned long b);
```

如果*a*（jiffies的某个快照）所代表的时间比*b*靠后，则第一个宏返回真；如果*a*比*b*靠前，则第二个宏返回真；后面两个宏分别用来比较“靠后或者相等”及“靠前或者相等”。这些宏会将计数器值转换为signed long，相减，然后比较结果。如果需要以安全的方式计算两个jiffies实例之间的差，也可以使用相同的技巧：

```
diff = (long)t2 - (long)t1;.
```

而通过下面的方法，可将两个jiffies的差转换为毫秒值：

```
msec = diff * 1000 / HZ;
```

但是，我们有时需要将来自用户空间的时间表述方法（使用struct timeval和struct timespec）和内核表述方法进行转换。这两个结构使用两个数来表示精确的时间：在老的、流行的struct timeval中使用秒和毫秒值，而较新的struct timespce中则使用秒和纳秒，前者比后者出现得早，但更常用。为了完成jiffies值和这些结构间的转换，内核提供了下面四个辅助函数：

```
#include <linux/time.h>

unsigned long timespec_to_jiffies(struct timespec *value);
void jiffies_to_timespec(unsigned long jiffies, struct timespec *value);
unsigned long timeval_to_jiffies(struct timeval *value);
void jiffies_to_timeval(unsigned long jiffies, struct timeval *value);
```

对64位jiffies_64的访问不像对jiffies的访问那样直接。在64位计算机架构上，这两个变量其实是同一个；但在32位处理器上，对64位值的访问不是原子的。这意味着，在我们读取64位值的高32位及低32位时，可能会发生更新，从而获得错误的值。因此，对64位计数器的直接读取是很靠不住的，但如果必须读取64位计数器，则应该使用内核导出的一个特殊辅助函数，该函数为我们完成了适当的锁定：

```
#include <linux/jiffies.h>
u64 get_jiffies_64(void);
```

在上面的函数原型中使用了u64类型。这是由*<linux/types.h>*定义的类型之一，我们将在第十一章讨论这些类型，它其实代表了一个无符号的64位类型。

如果读者对32位平台如何同时更新32位及64位计数器感到疑惑的话，可阅读对应平台上的链接器脚本（寻找其名称匹配于*vmlinux*.lds**的文件）。在链接器脚本中，jiffies符号被定义为访问64位值的高（或低）32位字，这取决于系统是大头的（big-endian）

还是小头的（little-endian）。实际上，相同的技巧也用于64位平台，这样，代码就会在同一地址访问unsigned long类型和u64类型的变量。

最后，需要注意，实际的时钟频率对用户空间来讲几乎是完全不可见的。当用户空间程序包含*param.h*时，HZ宏始终被扩展为100，而每个报告给用户空间的计数器值均做了相应的转换。这一说法适应于*clock(3)*、*times(2)*以及其他任何相关函数。对用户来讲，如果想知道定时器中断的确切HZ值，只能通过*/proc/interrupts*获得。例如，将通过*/proc/interrupts*获得的计数值除以*/proc/uptime*文件报告的系统运行时间，即可获得内核的确切HZ值。

处理器特定的寄存器

如果需要度量非常短的时间，或是需要极高的时间精度，就可以使用与特定平台相关的资源，这是将时间精度的重要性凌驾于代码的可移植性之上的做法。

在现代处理器中，由于缓存、指令调度、分支预测等技术的应用，在大部分的CPU设计中，指令时序本质上是不可预测的，这样，依赖于指令周期的经验型性能描述方法就不再适用。为了解决这一问题，CPU制造商引入了一种通过计算时钟周期来度量时间差的简便而可靠的方法，绝大多数现代处理器都包含一个随时钟周期不断递增的计数寄存器。这个时钟计数器是完成高分辨率计时任务的唯一可靠途径。

基于不同的平台，在用户空间，这个寄存器可能是可读的，也可能不可读；可能是可写的，也可能不可写；可能是64位的，也可能是32位的。如果是32位的，还得注意处理溢出的问题。在某些平台上，该寄存器可能根本不存在，或者如果CPU缺少这个特性，而我们又需要处理这种特殊的需求，则可能会由硬件设计者通过外部设备来实现。

无论该寄存器是否可以置0，我们都强烈建议不要重置它，即使硬件允许这么做。毕竟我们不是该计数器的唯一用户，例如在支持SMP的平台上，内核会依赖这种计数器来保持处理器之间的同步。因为总可以通过多次读取该寄存器并比较读出数值的差异来完成要做的事，故无需要求独占该寄存器并修改它的当前值。

最有名的计数器寄存器就是TSC（timestamp counter，时间戳计数器），从x86的Pentium处理器开始提供该寄存器，并包括在以后的所有CPU中，包括x86_64在内。它是一个64位的寄存器，记录CPU时钟周期数，从内核空间和用户空间都可以读取它。

包含头文件*<asm/msr.h>*（x86专用的头文件，意指“machine-specific registers，机器特有的寄存器”）之后，就可以使用如下的宏：

```
rdtsc(low32,high32);
```

```
rdtscl(low32);
rdtscll(var64);
```

第一个宏原子性地把64位的数值读到两个32位变量中；后一个只把寄存器的低半部分读入一个32位变量，而废弃高半部分；最后这个宏将64位值读入一个`long long`型的变量。上面所有的宏都会将值保存到对应的参数中。

在大多数常见的TSC应用中，读取计数器的低半部分就够了。1-GHz的处理器每4.2秒才会溢出，因此，如果我们度量的时间差确定很短的话，就不需要处理多个寄存器值。但是，随着CPU主频的提高以及计时需求的增加，将来肯定需要读取64位的计数值。

下面这段代码仅仅使用了该寄存器的低半部分，可用来测量该指令自身的运行时间：

```
unsigned long ini, end;
rdtscl(ini); rdtscl(end);
printk("time lapse: %li\n", end - ini);
```

其他一些平台也提供了类似的功能，在内核头文件中还有一个与体系结构无关的函数可以代替*rdtsc*，即*get_cycles*，它定义在*<asm/timex.h>*中（由*<linux/timex.h>*包含），其原型如下：

```
#include <linux/timex.h>
cycles_t get_cycles(void);
```

在各种平台上都可以使用这个函数，在没有时钟周期计数寄存器的平台上它总是返回0。`cycles_t`类型是能装入读取值的合适的无符号类型。

除了这个与体系结构无关的函数外，我们还将举例说明一段内嵌的汇编代码。为此，我们将针对MIPS处理器实现一个*rdtscl*函数，其功能与x86的一样。

这个例子之所以基于MIPS，是因为大多数MIPS处理器都有一个32位的计数器，在它们内部的“coprocessor 0”中称它为寄存器9。为了从内核空间读取该寄存器，可以定义下面的宏，它执行“从coprocessor 0读取”的汇编指令（注1）：

```
#define rdtscl(dest) \
   __asm__ __volatile__("mfc0 %0,$9; nop" : "=r" (dest))
```

通过使用这个宏，MIPS处理器就可以执行前面用于x86的代码了。

注1：尾部的*nop*指令是必需的，以防止编译器在指令*mfco*之后立即访问目标寄存器。这种互锁在RISC处理器中是很典型的，在延迟期间编译器仍然可以调度其他指令执行。我们在这里使用*nop*，是因为内嵌汇编对编译器而言是个黑盒子，不能进行优化。

gcc 内嵌汇编的有趣之处在于通用寄存器的分配使用是由编译器完成的。这个宏中使用的%0只是“参数0”的占位符，参数0由随后的“作为输出（=）使用的任意寄存器（r）”指定。该宏还声明了输出寄存器要对应于C的表达式dest。内嵌汇编的语法功能强大，但也比较复杂，特别是在对各寄存器使用有限制的平台上更是如此，如x86系列。完整的语法描述在 *gcc* 文档中提供，一般在 *info* 文档树中就可以找到。

本小节展示的短小的C代码段已经在一个K7系列的x86处理器和一个MIPS VR4181处理器（使用了刚才的宏）上运行过了。前者给出的时间消耗为11时钟周期，后者仅为2个时钟周期。这是可以理解的，因为RISC处理器通常在每时钟周期运行一条指令。

关于时间戳计数器，还有值得一提的一点：在SMP系统中，它们不会在多个处理器间保持同步。为了确保获得一致的值，我们需要为查询该计数器的代码禁止抢占。

获取当前时间

内核一般通过jiffies值来获取当前时间。该数值表示的是自最近一次系统启动到当前的时间间隔，它和设备驱动程序无关，因为它的生命期只限于系统的运行期（uptime）。但驱动程序可以利用jiffies的当前值来计算不同事件间的时间间隔（比如在输入设备驱动程序中就用它来分辨鼠标的单双击）。简而言之，利用jiffies值来测量时间间隔在大多数情况下已经足够了，如果还需要测量更短的时间差，就只能使用处理器特定的寄存器了（但这会带来严重的兼容性问题）。

驱动程序一般不需要知道墙钟时间（指日常生活使用的时间，用年月日来表达），通常只有像 *cron* 和 *syslogd* 这样用户程序才需要墙钟时间。对真实世界的时间处理通常最好留给用户空间，C函数库为我们提供了更好的支持。另外，这些代码通常具有更高的策略相关性，从而不能归于内核。但是，内核也提供了将墙钟时间转换为jiffies值的函数：

```
#include <linux/time.h>
unsigned long mktime (unsigned int year, unsigned int mon,
                      unsigned int day, unsigned int hour,
                      unsigned int min, unsigned int sec);
```

直接处理墙钟时间常常意味着正在实现某种策略，因此，我们应该仔细审视一下。

虽然在内核空间中我们不必处理时间的人类可读取表达，但有时也需要处理绝对时间戳。为此，*<linux/time.h>* 导出了 *do_gettimeofday* 函数。该函数用秒或微秒值来填充一个指向struct timeval的指针变量——*gettimeofday* 系统调用中用的也是同一变量。*do_gettimeofday* 的原型如下：

```
#include <linux/time.h>
void do_gettimeofday(struct timeval *tv);
```

此内核源代码表明*do_gettimeofday*在许多体系结构上有“接近微秒级的分辨率”，因为它通过查询定时硬件而得出了已经流逝在当前jiffies上的时间。但是，实际精度是随平台的不同而变化的，因为它依赖于实际使用的硬件机制。例如，某些*m68knommu*处理器、Sun3系统以及其他*m68k*系统无法提供高于jiffy的分辨率。另一方面，Pentium系统可通过读取本章前面描述的时间戳计数器来获得非常快而精确的子滴答度量值。

当前时间也可以通过xtime变量（类型为struct timespec）来获得，但精度要差一些。但是，我们并不鼓励直接使用该变量，因为很难原子地访问timeval变量的两个成员。因此，内核提供了一个辅助函数*current_kernel_time*:

```
#include <linux/time.h>
struct timespec current_kernel_time(void);
```

获取当前时间的代码可见于*jit*（“Just In Time”）模块中，其源文件可从O'Reilly公司的FTP站点获得。*jit*模块将创建*/proc/currentime*文件，读取该文件，将以ASCII码的形式返回下面几项数据:

- 以十六进制表达的jiffies及jiffies_64的当前值
- 由*do_gettimeofday*返回的当前时间
- 由*current_kernel_time*返回的timespec结构值

为了保持该示例模块的代码最少，我们选择了动态的*/proc*文件方式 —— 为了输出这几个不多的文本信息，不值得创建一个完整的设备。

在装载该模块之后，该文件就可以持续地返回文本行；每个*read*系统调用会收集并返回一组数据，为方便阅读，数据被组织为两行。如果在一次定时器滴答内多次读取的话，我们将看到*do_gettimeofday*返回值间的差异，因为它查询了相关硬件，而其他值只会在不同的定时器滴答间发生变化。

```
phon% head -8 /proc/currentime
0x00bdbc1f 0x0000000100bdbc1f 1062370899.630126
                              1062370899.629161488
0x00bdbc1f 0x0000000100bdbc1f 1062370899.630150
                              1062370899.629161488
0x00bdbc20 0x0000000100bdbc20 1062370899.630208
                              1062370899.630161336
0x00bdbc20 0x0000000100bdbc20 1062370899.630233
                              1062370899.630161336
```

在上面的输出中，有两个值得注意的现象。首先，尽管*current_kernel_time*以纳秒精度

表示，但只有时钟滴答的分辨率；*do_gettimeofday*持续报告靠后的时间，但总不会晚于下一个定时器滴答。其次，64位jiffies计数器的高32位字的最低位被值一，这是由于INTIAL_JIFFIES的默认值所致。在系统引导期间，这个值用来初始化jiffies计数器，这样，引导之后的几分钟之内就会出现低32位字的溢出，从而帮助检测每个溢出相关的问题。计数器上的这个初始值不会有任何效果，因为jiffies不是相对于墙钟时间的。在*/proc/uptime*中，当内核从计数器中抽取运行期时，会减去该初始值。

延迟执行

设备驱动程序经常需要将某些特定代码延迟一段时间后执行——通常是为了让硬件能完成某些任务。本节将介绍许多实现延迟的不同技术，哪种技术最好取决于实际环境中的具体情况。我们将介绍所有的这些技术并指出各自的优缺点。

要考虑的一件重要的事情是：相比时钟滴答并考虑各种平台上的HZ值范围，我们是否依赖于时钟滴答来实现延迟。长于时钟滴答的延迟并且不会因为它的低分辨率而导致问题，则可以使用系统时钟，而非常短的延迟通常必须用软件循环的方式实现。这两种情形之间存在一个灰色地带，本章我们把涉及多个时钟滴答的延迟称为“长延迟”，在某些平台上，这可能只有几个毫秒，但对CPU和内核来讲，仍然是比较长的了。

下面的几个小节讨论了许多不同的延迟方案，从直觉但并不合适的方案到正确的方案。我们选择这种讨论方式，是因为它能更深入讨论内核有关计时的问题。如果读者急于找到正确的代码，可直接跳过下面的小节。

长延迟

有时，驱动程序需要延迟比较长的时间，即长于一个时钟滴答。实现这种类型的延迟有好几种途径，我们首先讲述最简单的长延迟技术，然后再描述更高级一些的技术。

忙等待

如果想把执行延迟若干个时钟滴答，或者对延迟的精度要求不高，最简单（但我们并不推荐）的实现方法就是一个监视jiffies计数器的循环。这种忙等待的实现方法通常具有下面的形式，其中j1是延迟终止时的jiffies值：

```
while (time_before(jiffies, j1))
    cpu_relax();
```

对*cpu_relax*的调用将以架构相关的方式执行，其中不执行大量的处理器代码。在许多系统上，该函数根本不会做任何事情；而在对称多线程（“超线程”）系统上，它可能将

处理器让给其他线程。但不论是哪种情况，只要可能，我们都应该尽量避免使用这种方式。我们在这里提到它，只是因为读者可能偶尔需要运行这段代码，以便更好地理解其他的延迟技术。

还是先看看这段代码是如何工作的。因为内核的头文件中jiffies被声明为volatile型变量，所以每次C代码访问它时都会重新读取它，因此该循环可以起到延迟的作用。尽管也是“正确”的实现，但这个忙等待循环会严重降低系统性能。如果我们并没有将内核配置为抢占型的，那么这个循环将在延迟期间整个锁住处理器，而调度器从来不会抢占运行在内核空间中的进程，这样，在j1所代表的时间到来之前，计算机看起来就是死掉的。如果我们正在运行抢占式内核，则问题不会有这么严重，这是因为，除非代码拥有一个锁，否则处理器的时间还可以用作他用。但是在抢占式系统中，忙等待仍然有些浪费。

更糟糕的是，如果在进入循环之前正好禁止了中断，jiffies值就不会得到更新，那么while循环的条件就永远为真，这时，你不得不按下那只大的红按钮（指电源按钮）。

这种延迟和下面的几种延迟方法都在*jit*模块中实现了。由该模块创建的所有*/proc/jit**文件每次被读取一行文本（每行20字节）时都会延迟整整1秒。如果读者想测试忙等待代码，就可以读取*/proc/jitbusy*文件，它在返回每一行文本时都会进入忙等待循环并延迟1秒。

警告： 请确保每次从*/proc/jitbusy*中读取至多一行（或几行）。用来注册*/proc*文件的简化内核机制会反复调用*read*方法，以填充用户请求的数据缓冲区。因此，类似*cat/proc/jitbusy*这样的命令如果每次读取4KB，则会将计算机冻结205秒。

读取*/proc/jitbusy*的推荐命令是*dd bs=20 < /proc/jitbusy*，同时可以随意指定要读取的数据块大小。该文件返回每行有20字节的数据，它表示了延迟开始之前及之后的jiffies计数器值。下面是上述命令在某台低负荷计算机上的运行示例：

```
phon% dd bs=20 count=5 < /proc/jitbusy
  1686518   1687518
  1687519   1688519
  1688520   1689520
  1689520   1690520
  1690521   1691521
```

结果看起来很好：每个延迟都恰好是一秒（1000个jiffies），而下一个*read*系统调用会在前一个结束之后立即执行。但是，如果我们在运行有大量CPU密集型进程的系统（非抢占式内核）上运行时，会看到下面的输出：

```
phon% dd bs=20 count=5 < /proc/jitbusy
```

```
1911226   1912226
1913323   1914323
1919529   1920529
1925632   1926632
1931835   1932835
```

我们可以看到，每个 *read* 系统调用会恰好延迟 1 秒，但内核在调度 *dd* 进程执行下一个系统调用前可能要花费 5 秒的时间。对多任务系统来讲，这种现象是正常，因为CPU时间在所有运行的进程间共享，而CPU密集型进程具有动态降低的优先级（对调度策略的讨论已超出本书范围）。

上面在高负荷系统中的测试是运行 *load50* 示例程序时得到的。这个程序会 fork 大量进程，这些进程不做任何有效的事情但却大量消耗CPU资源。该程序是本书示例文件的一部分，默认会fork 50个进程，当然，具体的数字也可以通过命令行指定。在本章中，甚至是本书的其他地方，对高负荷系统的测试均是在相对空闲的计算机上运行*load50*来完成的。

如果在运行抢占式内核的高负荷系统上重复运行上面的命令，则会发现，和空闲CPU系统相比看不出任何大的区别，如下所示：

```
phon% dd bs=20 count=5 < /proc/jitbusy
 14940680  14942777
 14942778  14945430
 14945431  14948491
 14948492  14951960
 14951961  14955840
```

我们发现，在两次系统调用之间没有很大的延迟，但是单个延迟却可能长于 1 秒，甚至在上面的例子中达到 3.8 秒，而且会随时间的流逝而增加。这表明进程在其延迟过程中被中断，系统调度了其他进程。系统调用之间的空隙并不是调度该进程的唯一选择，因此我们没有看到系统调用之间出现特别延迟。

让出处理器

我们已经看到，忙等待为系统整体增加了沉重的负担，因此有必要寻找更好的延迟技术。我们能想到的一种手段是，在不需要 CPU 时主动释放 CPU。这可以通过调用 *schedule* 函数实现，该函数在 *<linux/sched.h>* 中声明：

```
while (time_before(jiffies, j1)) {
    schedule();
}
```

上面这个循环可通过读取 */proc/jitsched* 文件来测试，其过程和前面读取 */proc/jitbusy* 一样。但是，这仍然不是优化的技术。当前进程虽然释放了CPU而不做任何事情，但它仍

然在运行队列中。如果系统中只有一个可运行的进程，则该进程又会立即运行（它调用了调度器，而调度器选择了同一个进程，这个进程又调用调度器……）。换句话说，机器的负荷（运行进程的平均数）至少为一，而空闲（idle）任务（进程号为0，也因历史原因称为*swapper*）从来不会运行。尽管这个问题看起来似乎无关痛痒，但是，在计算机空闲时运行空闲任务可减轻处理器负荷、降低处理器温度并增加它的寿命，如果计算机是一台笔记本电脑，这种效果对笔记本电脑的电池也是一样的。此外，因为该进程在延迟期间真正在运行，因此它要对它消耗的所有时间负责。

在抢占式内核上，*/proc/jitsched*的行为和*/proc/jitbusy*的行为非常相似。在低负荷系统上，运行效果如下所示：

```
phon% dd bs=20 count=5 < /proc/jitsched
  1760205   1761207
  1761209   1762211
  1762212   1763212
  1763213   1764213
  1764214   1765217
```

我们可以注意到，*read*系统调用的实际延迟有时要比所请求的长几个时钟滴答。当系统越来越忙时，这个问题会越来越突出，最终会导致驱动程序等待更长的时间。当一个进程使用*schedule*释放处理器之后，没有任何保证说进程可以在随后很快就得到处理器。因此，除了影响计算系统的整体性能之外，上面这种调用*schedule*的方法对驱动程序需求来讲并不安全。如果在测试*jitseched*的同时运行*load50*，读者将看到每行的延迟可能达到好几秒，这是因为在延迟到期时其他进程正在使用CPU。

超时

到目前为止，通过监视jiffies计数器实现的延迟循环可以工作，但不是非常理想。读者可能想到，实现延迟的最好方法应该是让内核为我们完成相应工作。存在两种构造基于jiffies超时的途径，使用哪个则依赖于驱动程序是否在等待其他事件。

如果驱动程序使用等待队列来等待其他一些事件，而我们同时希望在特定时间段中运行，则可以使用*wait_event_timeout*或者*wait_event_interruptible_timeout*函数：

```
#include <linux/wait.h>
long wait_event_timeout(wait_queue_head_t q, condition, long timeout);
long wait_event_interruptible_timeout(wait_queue_head_t q,
                    condition, long timeout);
```

上述函数会在给定的等待队列上休眠，但是会在超时（用jiffies表示）到期时返回。这样，这两个函数实现了一种有界的休眠，这种休眠不会永远继续。注意，这里的timeout值表示的是要等待的jiffies值，而不是绝对时间值。这个值用有符号数表示，因为有些

情况下它是相减的结果。当提供的超时值是负数时，这两个函数会通过一条*printk*语句产生抱怨信息。如果超时到期，这两个函数会返回零；而如果进程由其他事件唤醒，则会返回剩余的延迟实现，并用jiffies表达。返回值从来不会是负数，即使因为系统负荷而导致真正的延迟时间超过预期。

*/proc/jitqueue*文件演示了基于*wait_event_interruptible_timeout*函数实现的延迟。因为这个模块并没有需要等待的事件，因此传入的条件（condition）是0：

```
wait_queue_head_t wait;
init_waitqueue_head (&wait);
wait_event_interruptible_timeout(wait, 0, delay);
```

根据观测结果，就算在高负荷系统上，对*/proc/jitqueue*的读取也将得到接近优化的结果：

```
phon% dd bs=20 count=5 < /proc/jitqueue
  2027024   2028024
  2028025   2029025
  2029026   2030026
  2030027   2031027
  2031028   2032028
```

因为读取进程（上面的*dd*）在等待超时的时候并不在运行队列中，因此无论是否在抢占式内核上运行上述代码，我们看不到任何区别。

在某个硬件驱动程序中使用*wait_event_timeout*和*wait_event_interruptible_timeout*时，执行的继续可通过下面两种方式获得：其他人在等待队列上调用了*wake_up*，或者超时到期。但这不适用于*jitqueue*，因为没有人会在等待队列上调用*wake_up*（毕竟没有其他代码知道这个等待队列），因此，进程始终会在超时到期时被唤醒。为了适应这种特殊情况（即不等待特定事件而延迟），内核为我们提供了*schedule_timeout*函数，这样，我们可避免声明和使用多余的等待队列头：

```
#include <linux/sched.h>
signed long schedule_timeout(signed long timeout);
```

这里，timeout是用jiffies表示的延迟时间。正常的返回值是0，除非在给定超时值到期前函数返回（比如响应某个信号）。*schedule_timeout*要求调用者首先设置当前进程的状态，因此，典型的调用代码如下所示：

```
set_current_state(TASK_INTERRUPTIBLE);
schedule_timeout (delay);
```

上面的两行语句（来自*/proc/jitschedto*）将使进程在给定时间内休眠。因为*wait_event_interruptible_timeout*在内部依赖于*schedule_timeout*函数，则*jitschedto*返

回的数值会和*jitqueue*一样，因此我们不打算给出*jitschedto*的输出。但是，值得再次指出的是，在超时到期和进程被真正调度执行之间，需要额外的时间。

在上面的示例中，第一行调用*set_current_state*以设置当前进程的状态，这样，调度器只会在超时到期且其状态变成TASK_RUNNING时才会运行这个进程。如果要实现不可中断的延迟，可使用TASK_UNINTERRUPTIBLE。如果我们忘记改变当前进程的状态，则对*schedule_timeout*的调用和对*schedule*的调用一样（即*jitsched*那样），内核为我们构造的定时器就不会真正起作用。

如果读者打算在不同的系统情况下或者不同的内核上，以不同的延迟执行方式测试上述四个*jit*文件，那么可以在装载模块时通过设置延迟模块的参数来配置具体的延迟时间值。

短延迟

当设备驱动程序需要处理硬件的延迟（latency）时，这种延迟通常最多涉及到几十个毫秒。在这种情况下，依赖于时钟滴答显然不是正确的方法。

ndelay、*udelay*和*mdelay*这几个内核函数可很好完成短延迟任务，它们分别延迟指定数量的纳秒、微秒和毫秒时间（注2）。它们的原型如下：

```
#include <linux/delay.h>
void ndelay(unsigned long nsecs);
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

这些函数的实际实现包含在*<asm/delay.h>*中，其实现和具体的体系架构相关，有时构建于一个外部函数。所有的体系架构都会实现*udely*，但其他函数可能未被定义；如果存在没有真正定义的函数，则*<linux/delay.h>*会在*udelay*的基础上提供一个默认的版本。不管哪种情况，真正实现的延迟至少会达到所请求的时间值，但可能更长；实际上，当前所有平台都无法达到纳秒精度，但有些平台提供了子微秒精度。延迟超过请求的值通常不是问题，因为驱动程序的短延迟通常等待的是硬件，而需求往往是至少要等待给定的时间段。

udelay（以及可能的*ndelay*）的实现使用了软件循环，它根据引导期间计算出的处理器速度以及loops_pre_jiffy整数变量确定循环的次数。如果读者要阅读实际的代码，要注意x86平台上的实现相当复杂，这是因为，它使用了不同的定时源，而这取决于运行代码的CPU类型。

注2：*udelay*中的u表示希腊字母“mu(μ)”，它代表“微（micro）”。

为避免循环计算中的整数溢出，*udelay*和*ndelay*为传递给它们的值强加了上限。如果模块无法装载，并显示未解析的符号 *__bad_udelay*，则说明模块在调用 *udelay* 时传入了太大的值。但需要注意的是，这种编译时的检查只能在常量值上进行，而且并不是所有的平台都实现了这种检查。作为一般性的规则，如果我们打算延迟上千个纳秒，则应该使用 *udelay* 而不是 *ndelay*；类似地，毫秒级的延迟也应该利用 *mdelay* 而不是更细粒度的短延迟函数。

要重点记住的是，这三个延迟函数均是忙等待函数，因而在延迟过程中无法运行其他任务。这样，这些函数将重复*jitbusy*的行为，只是在不同的量级上。因此，我们应该只在没有其他实用方法时使用这些函数。

实现毫秒级（或者更长）延迟还有另一种方法，这种方法不涉及忙等待。*<linux/delay.h>* 文件声明了下面这些函数：

```
void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int millisecs);
void ssleep(unsigned int seconds)
```

前两个函数将调用进程休眠以给定的 millisecs。对 *msleep* 的调用是不可中断的;我们可以确信进程将至少休眠给定的毫秒数。如果驱动程序正在某个等待队列上等待，而又希望有唤醒能够打断这个等待的话，则可使用 *msleep_interruptible*。*msleep_interruptible* 的返回值通常是零；但是，如果进程被提前唤醒，那么返回值就是原先请求休眠时间的剩余毫秒数。对 *ssleep* 的调用将使进程进入不可中断的休眠，但休眠时间以秒计。

通常，如果我们能够容忍比所请求更长的延迟，则应当使用 *chedule_timeout*、*sleep* 或者 *ssleep*。

内核定时器

如果我们需要在将来的某个时间点调度执行某个动作，同时在该时间点到达之前不会阻塞当前进程，则可以使用内核定时器。内核定时器可用来在未来的某个特定时间点（基于时钟滴答）调度执行某个函数，从而可用于完成许多任务；例如，如果硬件无法产生中断，则可以周期性地轮询设备状态。另一个内核定时器的典型应用是关闭软驱马达，或者结束其他长时间的关闭操作。在这种情况下，在 *close* 方法返回前进行延迟将会给应用程序带来不必要的（甚至令人惊讶的）开销。最后，内核本身也在许多情况下使用了定时器，包括在 *schedule_timeout* 的实现中。

一个内核定时器是一个数据结构，它告诉内核在用户定义的时间点使用用户定义的参数

来执行一个用户定义的函数。其实现位于*<linux/timer.h>*和*kernel/timer.c*文件，我们将在“内核定时器的实现”一节中对此进行详细描述。

被调度运行的函数几乎肯定不会在注册这些函数的进程正在执行时运行。相反，这些函数会异步地运行。到此为止，我们提供的示例驱动程序代码都在进程执行系统调用的上下文中运行。但是，当定时器运行时，调度该定时器的进程可能正在休眠或在其他处理器上执行，或干脆已经退出。

这种异步执行类似于硬件中断发生时的情景（我们会在第十章详细讨论）。实际上，内核定时器常常是作为“软件中断”的结果而运行的。在这种原子性的上下文中运行时，代码会受到许多限制。定时器函数必须以我们在第五章“自旋锁和原子上下文”一节中讨论的方式原子地运行，但是这种非进程上下文还带来其他一些问题。现在我们要讨论这些限制，这些限制还会在本书后面多次出现。我们也会对此多次重复，原子上下文中的这些规则必须遵守，否则会导致大麻烦。

许多动作需要在进程上下文中才能执行。如果处于进程上下文之外（比如在中断上下文中），则必须遵守如下规则：

- 不允许访问用户空间。因为没有进程上下文，无法将任何特定进程与用户空间关联起来。
- `current`指针在原子模式下是没有任何意义的，也是不可用的，因为相关代码和被中断的进程没有任何关联。
- 不能执行休眠或调度。原子代码不可以调用*schedule*或者*wait_event*，也不能调用任何可能引起休眠的函数。例如，调用*kmalloc(...,GFP_KERNEL)*就不符合本规则。信号量也不能用，因为可能引起休眠。

内核代码可以通过调用函数*in_interrupt()*来判断自己是否正运行于中断上下文，该函数无需参数，如果处理器运行在中断上下文就返回非零值，而无论是硬件中断还是软件中断。

和*in_interrupt()*相关的函数是*in_atomic()*。当调度不被允许时，后者的返回值也是非零值；调度不被允许的情况包括硬件和软件中断上下文以及拥有自旋锁的任何时间点。在后一种情况下，`current`是可用的，但禁止访问用户空间，因为这会导致调度的发生。不管何时使用*in_interrupt()*，都应考虑是否真正该使用的是*in_atomic()*。这两个函数均在*<asm/hardirq.h>*中声明。

内核定时器的另一个重要特性是，任务可以将自己注册以在稍后的时间重新运行。这种可能性是因为每个`timer_list`结构都会在运行之前从活动定时器链表中移走，这样

就可以立即链入其他的链表。尽管多次调度同一任务似乎是一件没有多大意义的操作，但有时还是很有用的。例如，这种技术可在轮询设备时使用。

另外一个值得了解的是，在SMP系统中，定时器函数会由注册它的同一CPU执行，这样可以尽可能获得缓存的局域性（locality）。因此，一个注册自己的定时器始终会在同一CPU上运行。

关于定时器，还有一个要谨记的重要特性：即使在单处理器系统上，定时器也会是竞态的潜在来源。这是由其异步执行的特点直接导致的。因此，任何通过定时器函数访问的数据结构都应该针对并发访问进行保护，可以使用第五章“原子变量”一节中讨论过的原子类型，或者使用第五章中讨论过的自旋锁。

定时器API

内核为驱动程序提供了一组用来声明、注册和删除内核定时器的函数。下面摘录了一些基本的接口：

```
#include <linux/timer.h>
struct timer_list {
        /* ... */
        unsigned long expires;
        void (*function)(unsigned long);
        unsigned long data;
};

void init_timer(struct timer_list *timer);
struct timer_list TIMER_INITIALIZER(_function, _expires, _data);

void add_timer(struct timer_list * timer);
int del_timer(struct timer_list * timer);
```

上面给出的数据结构其实包含其他一些未列出的字段，但给出的三个字段是可由定时器代码以外的代码访问。expires字段表示期望定时器执行的jiffies值；到达该jiffies值时，将调用*function*函数，并传递data作为参数。如果需要通过这个参数传递多个数据项，那么可以将这些数据项捆绑成一个数据结构，然后将该数据结构的指针强制转换成unsigned long传入。这种技巧在所有内核支持的体系架构上都是安全的，而且在内存管理（参见第十五章的讨论）中非常常见。expires的值并不是jiffies_64项，这是因为定时器并不适用于长的未来时间点，而且32位平台上的64位操作会比较慢。

该结构在使用前必须初始化。初始化步骤可确保所有的字段被正确设置，包括那些对调用者不可见的字段。通过调用*init_timer*或者将TIMER_INITIALIZER赋予某个静态的结

构，即可完成初始化，使用哪个方法取决于我们自己的需求。在初始化之后，可在调用*add_timer*之前修改上面讲到的三个公共字段。如果要在定时器到期前禁止一个已注册的定时器，则可以调用*del_timer*函数。

*jit*模块包含一个示例文件，即*/proc/jitimer*（表示“just in timer”），该文件返回一个标题行以及6个数据行。数据行表示的是代码运行时的当前环境；第一行由*read*文件操作生成，而其他的行由定时器生成。下面的输出是正在编译内核时得到的：

```
phon% cat /proc/jitimer
   time   delta  inirq    pid   cpu command
 33565837     0      0     1269   0   cat
 33565847    10      1     1271   0   sh
 33565857    10      1     1273   0   cpp0
 33565867    10      1     1273   0   cpp0
 33565877    10      1     1274   0   cc1
 33565887    10      1     1274   0   cc1
```

在上面的输出中，time字段是代码运行时的jiffies值，delta是自前一行以来jiffies的变化值；inirq是由*in_interrupt*返回的布尔值，pid和command表示当前进程，而cpu是正在使用的CPU编号（在单处理器系统上始终为0）。

如果在系统低负荷时读取*/proc/jitimer*，将发现定时器的上下文会是进程0，即空闲任务，该任务因历史原因而被称为“swapper”。

用来生成*/proc/jitimer*数据的定时器默认情况下每10个jiffies运行一次，但读者可在装载该模块时通过设置tdelay（timer delay，定时器延迟）参数来修改这个值。

下面是*jit*模块中和*jitimer*定时器相关的代码。当某个进程试图读取*/proc/jitimer*文件时，设置定时器如下：

```
unsigned long j = jiffies;

/* 为定时器函数填充数据 */
data->prevjiffies = j;
data->buf = buf2;
data->loops = JIT_ASYNC_LOOPS;

/* register the timer */
data->timer.data = (unsigned long)data;
data->timer.function = jit_timer_fn;
data->timer.expires = j + tdelay; /* 参数 */
add_timer(&data->timer);

/* 等待缓冲区以填充 */
wait_event_interruptible(data->wait, !data->loops);
```

实际的定时器函数如下：

```
void jit_timer_fn(unsigned long arg)
{
    struct jit_data *data = (struct jit_data *)arg;
    unsigned long j = jiffies;
    data->buf += sprintf(data->buf, "%9li  %3li     %i    %6i   %i   %s\n",
                 j, j - data->prevjiffies, in_interrupt() ? 1 : 0,
                 current->pid, smp_processor_id(), current->comm);

    if (--data->loops) {
        data->timer.expires += tdelay;
        data->prevjiffies = j;
        add_timer(&data->timer);
    } else {
        wake_up_interruptible(&data->wait);
    }
}
```

除了上面给出的函数及接口以外，内核定时器API还包括其他几个函数。下面给出这些函数的完整描述：

`int mod_timer(struct timer_list *timer, unsigned long expires);`

更新某个定时器的到期时间，经常用于超时定时器（典型的例子是软驱的关马达定时器）。我们也可以在通常使用*add_timer*的时候在不活动的定时器上调用*mod_timer*。

`int del_timer_sync(struct timer_list *timer);`

和*del_timer*的工作类似，但该函数可确保在返回时没有任何CPU在运行定时器函数。*del_timer_sync*可用于在SMP系统上避免竞态，这和单处理器内核中的*del_timer*是一样的。在大多数情况下，应优先考虑调用这个函数而不是*del_timer*函数。如果从非原子上下文调用，该函数可能休眠，但在其他情况下会进入忙等待。在拥有锁时，应格外小心调用*del_timer_sync*，因为如果定时器函数企图获取相同的锁，系统就会进入死锁。如果定时器函数会重新注册自己，则调用者必须首先确保不会发生重新注册；这通常通过设置一个由定时器函数检查的“关闭”标志来实现。

`int timer_pending(const struct timer_list * timer);`

该函数通过读取timer_list结构的一个不可见字段来返回定时器是否正在被调度运行。

内核定时器的实现

尽管要使用内核定时器并不必知道它们的具体实现，但其实现非常有意思，而了解其内部也是值得的。

内核定时器的实现要满足如下需求及假定:

- 定时器的管理必须尽可能做到轻量级。
- 其设计必须在活动定时器大量增加时具有很好的伸缩性。
- 大部分定时器会在最多几秒或者几分钟内到期，而很少存在长期延迟的定时器。
- 定时器应该在注册它的同一CPU上运行。

内核开发者使用的解决方案是利用per-CPU数据结构。*timer_list*结构的base字段包含了指向该结构的指针。如果base为NULL，定时器尚未调度运行；否则，该指针会告诉我们哪个数据结构（也就是哪个CPU）在运行定时器。Per-CPU数据项在第八章的“Per-CPU变量”一节中描述。

不管何时内核代码注册了一个定时器（通过*add_tiemr*或者*mod_timer*），其操作最终会由*internal_add_timer*（定义在kernel/timer.c中）执行，该函数又会将新的定时器添加到和当前CPU关联的“级联表”中的定时器双向链表中。

级联表的工作方式如下：如果定时器在接下来的0～255个jiffies中到期，则该定时器就会被添加到256个链表中的一个（这取决于expires字段的低8位值），这些链表专用于短期定时器。如果定时器会在较远的未来到期（但在16384个jiffies之前），则该定时器会被添加到64个链表之一（这取决于expires字段的9～14位）。对更远将来的定时器，相同的技巧用于15～20位、21～26位以及27～31位。如果定时器的expires字段代表了更远的未来（只可能发生在64位系统上），则利用延迟值0xffffffff做散列（hash）运算，而在过去时间内到期的定时器会在下一个定时器滴答时被调度（在高负荷的情况下，有可能注册一个已经到期的定时器，尤其在运行抢占式内核时）。

当*__run_timers*被激发时，它会执行当前定时器滴答上的所有挂起的定时器。如果jiffies当前是256的倍数，该函数还会将下一级定时器链表重新散列到256个短期链表中，同时还可能根据上面jiffies的位划分对将其他级别的定时器做级联处理。

这种方法虽然初看起来有些复杂，但能很好地处理定时器不多或有大量定时器的情况。用来管理每个活动定时器所需的必要时间和已注册的定时器数量无关，同时被限于定时器expires字段二进制表达上的几个逻辑操作。这种实现唯一的开销在于512个链表头（256个短期链表以及4组64个的长期链表）占用了4KB的存储空间。

如同*/proc/jitimer*所描述的，函数*__run_timers*运行在原子上下文中。除了我们已经描述过的限制外，这带来了一个有趣的特点：定时器会在正确的时间到期，即使我们运行的不是抢占式的内核，而CPU会忙于内核空间。如果读者在后台读取*/proc/jitbusy*而在

前台读取*/etc/jitimer*时，就能看到这个特点。尽管系统似乎被忙等待系统调用整个锁住，但内核定时器仍然可很好地工作。

但需要谨记的是，内核定时器离完美还有很大距离，因为它受到jitter以及由硬件中断、其他定时器和异步任务所产生的影响。和简单数字I/O关联的定时器对简单任务来说足够了，比如控制步进电机或者业余电子设备，但通常不适合于工业环境下的生产系统。对这类任务，我们需要借助某种实时的内核扩展。

tasklet

和定时问题相关的另一个内核设施是tasklet（小任务）机制。中断管理（第十章将进一步描述）中大量使用了这种机制。

tasklet在很多方面类似内核定时器：它们始终在中断期间运行，始终会在调度它们的同一CPU上运行，而且都接收一个`unsigned long`参数。和内核定时器不同的是，我们不能要求tasklet在某个给定时间执行。调度一个tasklet，表明我们只是希望内核选择某个其后的时间来执行给定的函数。这种行为对中断处理例程来说尤其有用，中断处理例程必须尽可能快地管理硬件中断，而大部分数据管理则可以安全地延迟到其后的时间。实际上，和内核定时器类似，tasklet也会在“软件中断”上下文以原子模式执行。软件中断是打开硬件中断的同时执行某些异步任务的一种内核机制。

tasklet以数据结构的形式存在，并在使用前必须初始化。调用特定的函数或者使用特定的宏来声明该结构，即可完成tasklet的初始化：

```
#include <linux/interrupt.h>

struct tasklet_struct {
      /* ... */
      void (*func)(unsigned long);
      unsigned long data;
};

void tasklet_init(struct tasklet_struct *t,
      void (*func)(unsigned long), unsigned long data);
DECLARE_TASKLET(name, func, data);
DECLARE_TASKLET_DISABLED(name, func, data);
```

tasklet为我们提供了许多有意思的特性：

- 一个tasklet可在稍后被禁止或者重新启用；只有启用的次数和禁止的次数相同时，tasklet才会被执行。
- 和定时器类似，tasklet可以注册自己本身。

- tasklet可被调度以在通常的优先级或者高优先级执行。高优先级的tasklet总会首先执行。
- 如果系统负荷不重，则tasklet会立即得到执行，但始终不会晚于下一个定时器滴答。
- 一个tasklet可以和其他tasklet并发，但对自身来讲是严格串行处理的，也就是说，同一tasklet永远不会在多个处理器上同时运行。当然我们已经指出，tasklet始终会在调度自己的同一CPU上运行。

*jit*模块包含两个文件，即*/proc/jitasklet*和*/proc/jitaskethi*，它们返回的数据和“内核定时器”一节中讲到的*/proc/jitimer*相同。在读取其中一个文件时，我们将获得一个标题行和6个数据行。第一个数据行描述了调用进程的上下文，而其他行则描述了其后运行tasklet时的上下文。当编译内核时运行下面的命令，将给出如下输出：

```
phon% cat /proc/jitasklet
   time   delta  inirq    pid   cpu command
 6076139     0      0     4370   0   cat
 6076140     1      1     4368   0   cc1
 6076141     1      1     4368   0   cc1
 6076141     0      1        2   0   ksoftirqd/0
 6076141     0      1        2   0   ksoftirqd/0
 6076141     0      1        2   0   ksoftirqd/0
```

如上面数据所证实，只要CPU忙于运行某个进程，tasklet就会在下一个定时器滴答运行，但如果CPU空闲，则会立即运行。内核为每个CPU提供了一组*ksoftirq*内核线程，用于运行“软件中断”处理例程，比如*tasklet_action*函数。这样，最后三次tasklet的运行都发生在与0号CPU关联的*ksoftirqd*内核线程的上下文中。*jitasklethi*的实现使用了高优先级的tasklet，下面将解释相关的函数。

用来实现*/proc/jitasklet*和*/proc/jitasklethi*的*jit*模块代码几乎和实现*/proc/jitimer*的代码一模一样，只是前者使用了tasklet调用而不是定时器接口。下面的清单描述了tasklet相关的内核接口，可在tasklet结构被初始化之后使用：

```
void tasklet_disable(struct tasklet_struct *t);
```

这个函数禁用指定的tasklet。该tasklet仍然可以用*tasklet_schedule*调度，但其执行被推迟，直到该tasklet被重新启用。如果tasklet当前正在运行，该函数会进入忙等待直到tasklet退出为止；因此，在调用*tasklet_disable*之后，我们可以确信该tasklet不会在系统中的任何地方运行。

```
void tasklet_disable_nosync(struct tasklet_struct *t);
```

禁用指定的tasklet，但不会等待任何正在运行的tasklet退出。该函数返回后，tasklet

是禁用的，而且在重新启用之前，不会再次被调度。但是，当该函数返回时，指定的tasklet可能仍在其他CPU上运行。

`void tasklet_enable(struct tasklet_struct *t);`

启用一个先前被禁用的tasklet。如果该tastlet已经被调度，它很快就会运行。对*tasklet_enable*的调用必须和每个对*tasklet_disable*的调用匹配，因为内核对每个tasklet保存有一个“禁用计数”。

`void tasklet_schedule(struct tasklet_struct *t);`

调度执行指定的tasklet。如果在获得运行机会之前，某个tasklet被再次调度，则该tasklet只会运行一次。但是如果在该tasklet运行时被调度，就会在完成后再次运行。这样，可确保正在处理事件时发生的其他事件也会被接收并注意到。这种行为也允许tasklet重新调度自身。

`void tasklet_hi_schedule(struct tasklet_struct *t);`

调度指定的tasklet以高优先级执行。当软件中断处理例程运行时，它会在处理其他软件中断任务（包括“通常”的tasklet）之前处理高优先级的tasklet。理想状态下，只有具备低延迟需求的任务（比如填充音频缓冲区）才能使用这个函数，这样可避免由其他软件中断处理例程引入的额外延迟。和*/proc/jitasklet*相比，*/proc/jitasklethi*给出了肉眼能察觉的区别。

`void tasklet_kill(struct tasklet_struct *t);`

该函数确保指定的tasklet不会被再次调度运行；当设备要被关闭或者模块要被移除时，我们通常调用这个函数。如果tasklet正被调度执行，该函数会等待其退出。如果tasklet重新调度自己，则应该避免在调用*tasklet_kill*之前完成重新调度，这和*del_timer_sync*的处理类似。

tasklet的实现在*kernel/softirq.c*中。其中有两个（通常优先级和高优先级的）tasklet链表，它们作为per-CPU数据结构而声明，并且使用了类似内核定时器那样的CPU相关机制。tasklet管理中使用的数据结构是个简单的链表，因为tasklet不必像内核定时器那样来处理时间问题。

工作队列

从表面看来，工作队列（workqueue）类似于tasklet，它们都允许内核代码请求某个函数在将来的时间被调用。但是，两者之间存在一些非常重要的区别，其中包括：

- tasklet在软件中断上下文中运行，因此，所有的tasklet代码都必须是原子的。相反，工作队列函数在一个特殊内核进程的上下文中运行，因此它们具有更好的灵活性。尤其是，工作队列函数可以休眠。

- tasklet始终运行在被初始提交的同一处理器上，但这只是工作队列的默认方式。
- 内核代码可以请求工作队列函数的执行延迟给定的时间间隔。

两者的关键区别在于：tasklet会在很短的时间段内很快执行，并且以原子模式执行、而工作队列函数可具有更长的延迟并且不必原子化。两种机制有各自适合的情形。

工作队列有struct workqueue_struct的类型，该结构定义在<*linux/workqueue.h*>中。在使用之前，我们必须显式地创建一个工作队列，可使用下面两个函数之一：

```
struct workqueue_struct *create_workqueue(const char *name);
struct workqueue_struct *create_singlethread_workqueue(const char *name);
```

每个工作队列有一个或多个专用的进程（“内核线程”），这些进程运行提交到该队列的函数。如果我们使用*create_workqueue*，则内核会在系统中的每个处理器上为该工作队列创建专用的线程。在许多情况下，众多的线程可能对性能具有某种程度的杀伤力；因此，如果单个工作线程足够使用，那么应该使用*create_singlethread_workqueue*创建工作队列。

要向一个工作队列提交一个任务，需要填充一个work_struct结构，这可通过下面的宏在编译时完成：

```
DECLARE_WORK(name, void (*function)(void *), void *data);
```

其中，name是要声明的结构名称，function是要从工作队列中调用的函数，而data是要传递给该函数的值。如果要在运行时构造work_struct结构，可使用下面两个宏：

```
INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);
PREPARE_WORK(struct work_struct *work, void (*function)(void *), void *data);
```

*INIT_WORK*完成更加彻底的结构初始化工作；在首次构造该结构时，应该使用这个宏。*PREPARE_WORK*完成几乎相同的工作，但它不会初始化用来将work_struct结构链接到工作队列的指针。如果结构已经被提交到工作队列，而只是需要修改该结构，则应该使用*PREPARE_WORK*而不是*INIT_WORK*。

如果要将工作提交到工作队列，则可使用下面的两个函数之一：

```
int queue_work(struct workqueue_struct *queue, struct work_struct *work);
int queue_delayed_work(struct workqueue_struct *queue,
                       struct work_struct *work, unsigned long delay);
```

它们都会将work添加到给定的queue。但是如果使用*queue_delayed_work*，则实际的工作至少会在经过指定的jiffies（由delay指定）之后才会被执行。如果工作被成功添

加到队列，则上述函数的返回值为1。返回值为非零时意味着给定的work_struct结构已经等待在该队列中，从而不能两次加入该队列。

在将来的某个时间，工作函数会被调用，并传入给定的data值。该函数会在工作线程的上下文运行，因此如果必要，它可以休眠——当然，我们应该仔细考虑休眠会不会影响提交到同一工作队列的其他任务。但是该函数不能访问用户空间，这是因为它运行在内核线程，而该线程没有对应的用户空间可以访问。

如果要取消某个挂起的工作队列入口项，可调用：

```
int cancel_delayed_work(struct work_struct *work);
```

如果该入口项在开始执行前被取消，则上述函数返回非零值。在调用*cancel_delayed_work*之后，内核会确保不会初始化给定入口项的执行。但是，如果*cancel_delayed_work*返回0，则说明该入口项已经在其他处理器上运行，因此在*cancel_delayed_work*返回后可能仍在运行。为了绝对确保在*cancel_delayed_work*返回0之后，工作函数不会在系统中的任何地方运行，则应该随后调用下面的函数：

```
void flush_workqueue(struct workqueue_struct *queue);
```

在*flush_workqueue*返回后，任何在该调用之前被提交的工作函数都不会在系统任何地方运行。

在结束对工作队列的使用后，可调用下面的函数释放相关资源：

```
void destroy_workqueue(struct workqueue_struct *queue);
```

共享队列

在许多情况下，设备驱动程序不需要有自己的工作队列。如果我们只是偶尔需要向队列中提交任务，则一种更简单、更有效的办法是使用内核提供的共享的默认工作队列。但是，如果我们使用这个工作队列，则应该记住我们正在和其他人共享该工作队列。这意味着，我们不应该长期独占该队列，即不能长时间休眠，而且我们的任务可能需要更长的时间才能获得处理器时间。

jiq（“just in queue”）模块导出了两个文件，这两个文件演示了共享队列的使用。它们使用了单个work_struct结构，并用下面的方式进行初始化：

```
static struct work_struct jiq_work;
    /* 这行语句出现在 jiq_init() 中 */
    INIT_WORK(&jiq_work, jiq_print_wq, &jiq_data);
```

当进程读取*/proc/jiqwq*时，该模块会通过共享工作队列初始化一系列的trip，并不作任何延迟。它使用的函数是：

```
int schedule_work(struct work_struct *work);
```

注意，在利用共享队列时，模块使用的是不同的函数，因为它要求work_struct结构能够带有参数。*jiq*中的实际代码如下：

```
prepare_to_wait(&jiq_wait, &wait, TASK_INTERRUPTIBLE);
schedule_work(&jiq_work);
schedule();
finish_wait(&jiq_wait, &wait);
```

实际的工作函数打印了一行文本，这类似*jit*模块，然后如有必要，将work_struct结构重新提交给工作队列。下面是完整的*jiq_print_wq*函数：

```
static void jiq_print_wq(void *ptr)
{
    struct clientdata *data = (struct clientdata *) ptr;

    if (! jiq_print (ptr))
        return;

    if (data->delay)
        schedule_delayed_work(&jiq_work, data->delay);
    else
        schedule_work(&jiq_work);
}
```

如果用户读取延迟的设备（*/proc/jjiqwqdelay*），工作函数会将自己以延迟模式重新提交到工作队列，这时使用*schedule_delayed_work*函数：

```
int schedule_delayed_work(struct work_struct *work, unsigned long delay);
```

如果查看这两个设备上的输出，其结果如下所示：

```
% cat /proc/jiqwq
    time  delta preempt   pid cpu command
  1113043     0       0     7   1 events/1
  1113043     0       0     7   1 events/1
  1113043     0       0     7   1 events/1
  1113043     0       0     7   1 events/1
  1113043     0       0     7   1 events/1
% cat /proc/jiqwqdelay
    time  delta preempt   pid cpu command
  1122066     1       0     6   0 events/0
  1122067     1       0     6   0 events/0
  1122068     1       0     6   0 events/0
  1122069     1       0     6   0 events/0
  1122070     1       0     6   0 events/0
```

读取 */proc/jiqwq* 时，每行的打印看不到明显的延迟。相反，读取 */proc/jiqwqdelay* 时，在每行之间存在明显的一个jiffies延迟。不论哪种情况，我们都能看到名为printed的同一进程，它就是实现共享工作队列的内核线程。CPU编号打印在斜杠之后，我们无法知道读取 */proc* 文件时究竟运行在哪个CPU上，但工作函数其后会始终运行在同一处理器上。

如果需要取消已提交到共享队列中的工作入口项，则可使用上面描述过的 *cancel_delayed_work* 函数。但是，刷新共享工作队列时需要另一个函数：

```
void flush_scheduled_work(void);
```

因为我们无法知道其他人是否在使用该队列，因此我们也无法知道在 *flush_scheduled_work* 返回前到底要花费多少时间。

快速参考

本章引入了如下符号：

计时

```
#include <linux/param.h>
HZ
```

HZ 符号指出每秒钟产生的时钟滴答数。

```
#include <linux/jiffies.h>
volatile unsigned long jiffies
u64 jiffies_64
```

jiffies_64 变量会在每个时钟滴答递增，也就是说，它会在每秒递增 HZ 次。内核代码大部分情况下使用 jiffies，在64位平台上，它和 jiffies_64 是一样的，而在 32 位平台上，jiffies 是 jiffies_64 的低 32 位。

```
int time_after(unsigned long a, unsigned long b);
int time_before(unsigned long a, unsigned long b);
int time_after_eq(unsigned long a, unsigned long b);
int time_before_eq(unsigned long a, unsigned long b);
```

这些布尔表达式以安全方式比较jiffies，无需考虑计数器溢出的问题，也不必访问 jiffies_64。

```
u64 get_jiffies_64(void);
```

无竞态地获取 jiffies_64 的值。

```
#include <linux/time.h>
unsigned long timespec_to_jiffies(struct timespec *value);
void jiffies_to_timespec(unsigned long jiffies, struct timespec *value);
unsigned long timeval_to_jiffies(struct timeval *value);
void jiffies_to_timeval(unsigned long jiffies, struct timeval *value);
```

在jiffies表示的时间和其他表示法之间转换。

```
#include <asm/msr.h>
rdtsc(low32,high32);
rdtscl(low32);
rdtscll(var32);
```

x86专用的宏，用来读取时间戳计数器。上述宏用两个32位字的形式读取该计数器，要么读取低32位，要么整个读取到一个long long型的变量中。

```
#include <linux/timex.h>
cycles_t get_cycles(void);
```

以平台无关的方式返回时间戳计数器。如果CPU不提供时间戳特性，则返回0。

```
#include <linux/time.h>
unsigned long mktime(year, mon, day, h, m, s);
```

根据6个无符号的int参数返回自Epoch以来的秒数。

```
void do_gettimeofday(struct timeval *tv);
```

以自Epoch以来的秒数和毫秒数的形式返回当前时间，并且以硬件能提供的最好分辨率返回。在大多数平台上，分辨率是微秒或更好，但某些平台只能提供jiffies级的分辨率。

```
struct timespec current_kernel_time(void);
```

以jiffies为分辨率返回当前时间。

延迟

```
#include <linux/wait.h>
long wait_event_interruptible_timeout(wait_queue_head_t *q, condition, signed
  long timeout);
```

使当前进程休眠在等待队列上，并指定用jiffies表达的超时值。如果要进入不可中断休眠，则应使用 *schedule_timeout*（见下）。

```
#include <linux/sched.h>
signed long schedule_timeout(signed long timeout);
```

调用调度器，确保当前进程可在给定的超时值之后被唤醒。调用者必须首先调用*set_current_state*将自己置于可中断或不可中断的休眠状态。

```
#include <linux/delay.h>
void ndelay(unsigned long nsecs);
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

引入整数的纳秒、微秒和毫秒级延迟。实际达到的延迟至少是请求的值，但可能更长。传入每个函数的参数不能超过平台相关的限制（通常是几千）。

```
void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int millisecs);
void ssleep(unsigned int seconds);
```

使进程休眠给定的毫秒数（或使用*ssleep*休眠给定的秒数）。

内核定时器

```
#include <asm/hardirq.h>
int in_interrupt(void);
int in_atomic(void);
```

返回布尔值以告知调用代码是否在中断上下文或者在原子上下文中执行。中断上下文在进程上下文之外，可能正处理硬件或软件中断。原子上下文是指不能进行调度的时间点，比如中断上下文或者拥有自旋锁时的进程上下文。

```
#include <linux/timer.h>
void init_timer(struct timer_list * timer);
struct timer_list TIMER_INITIALIZER(_function, _expires, _data);
```

上面的函数以及静态声明定时器结构的宏是初始化timer_list数据结构的两种方式。

```
void add_timer(struct timer_list * timer);
```

注册定时器结构，以在当前CPU上运行。

```
int mod_timer(struct timer_list *timer, unsigned long expires);
```

修改一个已经调度的定时器结构的到期时间。它也可以替代add_timer函数使用。

```
int timer_pending(struct timer_list * timer);
```

返回布尔值的宏，用来判断给定的定时器结构是否已经被注册运行。

```
void del_timer(struct timer_list * timer);
void del_timer_sync(struct timer_list * timer);
```

从活动定时器清单中删除一个定时器。后一个函数确保定时器不会在其他CPU上运行。

tasklet

```
#include <linux/interrupt.h>
DECLARE_TASKLET(name, func, data);
DECLARE_TASKLET_DISABLED(name, func, data);
void tasklet_init(struct tasklet_struct *t, void (*func)(unsigned long),
  unsigned long data);
```

前面两个宏声明一个tasklet结构，而*tasklet_init*函数初始化一个通过分配或者其他途径获得的tasklet结构。第二个DESCLARE宏禁用给定的tasklet。

```
void tasklet_disable(struct tasklet_struct *t);
void tasklet_disable_nosync(struct tasklet_struct *t);
void tasklet_enable(struct tasklet_struct *t);
```

禁用或重新启用某个tasklet。每次禁止都要匹配一次使能（我们可以禁用一个已经被禁用的tasklet）。*tasklet_disable*函数会在tasklet正在其他CPU上运行时等待，而*nosync*版本不会完成这个额外的步骤。

```
void tasklet_schedule(struct tasklet_struct *t);
void tasklet_hi_schedule(struct tasklet_struct *t);
```

调度运行某个tasklet，可以是“通常”的tasklet或者一个高优先级的tasklet。当执行软件中断时，高优先级的tasklet会被首先处理，而通常的tasklet最后运行。

```
void tasklet_kill(struct tasklet_struct *t);
```

如果指定的tasklet被调度运行，则将其从活动链表中删除。和*tasklet_disable*类似，该函数可在SMP系统上阻塞，以便等待正在其他CPU上运行的该tasklet终止。

工作队列

```
#include <linux/workqueue.h>
struct workqueue_struct;
struct work_struct;
```

上述结构分别表示工作队列和工作入口项。

```
struct workqueue_struct *create_workqueue(const char *name);
struct workqueue_struct *create_singlethread_workqueue(const char *name);
void destroy_workqueue(struct workqueue_struct *queue);
```

用于创建和销毁工作队列的函数。调用*create_workqueue*将创建一个队列，且系统中的每个处理器上都会运行一个工作线程；相反，*create_singlethread_workqueue*只会创建单个工作进程。

```
DECLARE_WORK(name, void (*function)(void *), void *data);
INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);
PREPARE_WORK(struct work_struct *work, void (*function)(void *), void *data);
```

用于声明和初始化工作队列入口项的宏。

```
int queue_work(struct workqueue_struct *queue, struct work_struct *work);
int queue_delayed_work(struct workqueue_struct *queue, struct
  work_struct *work, unsigned long delay);
```

用来安排工作以便从工作队列中执行的函数。

```
int cancel_delayed_work(struct work_struct *work);
void flush_workqueue(struct workqueue_struct *queue);
```

使用*cancel_delayed_work*可从工作队列中删除一个入口项；*flush_workqueue*确保系统中任何地方都不会运行任何工作队列入口项。

```
int schedule_work(struct work_struct *work);
int schedule_delayed_work(struct work_struct *work, unsigned long delay);
void flush_scheduled_work(void);
```

使用共享工作队列的函数。



第八章

分配内存

到目前为止，我们已经使用过*kmalloc*和*kfree*来分配和释放内存，但Linux内核实际上提供了更加丰富的内存分配原语集。本章我们将会介绍设备驱动程序中使用内存的一些其他方法，还会介绍如何最好地利用系统内存资源。我们不会讨论不同体系结构是如何实际管理内存的。因为内核为设备驱动程序提供了一致的内存管理接口，所以模块不需要涉及分段、分页等问题。另外，本章也不会描述内存管理的内部细节，这些问题将留到第十五章讨论。

kmalloc函数的内幕

*kmalloc*内存分配引擎是一个功能强大的工具，由于和*malloc*相似，所以学习它也很容易。除非被阻塞，否则这个函数可运行得很快，而且不对所获取的内存空间清零，也就是说，分配给它的区域仍然保持着原有的数据（注1）。它分配的区域在物理内存中也是连续的。在下面几节中，我们将详细讨论*kmalloc*函数，读者可以把它和后面将要讨论的其他一些内存分配技术做个比较。

flags参数

记住*kmalloc*的原型是：

```
#include <linux/slab.h>

void *kmalloc(size_t size, int flags);
```

注1：最重要的是，这意味着我们要将内存显式地清空，尤其是可能导出给用户空间或者写入设备的内存；否则，就可能将私有信息泄露出去。

*kmalloc*的第一个参数是要分配的块的大小，第二个参数是分配标志（flags），更有意思的是，它能够以多种方式控制*kmalloc*的行为。

最常用的标志是GFP_KERNEL，它表示内存分配（最终总是调用*get_free_pages*来实现实际的分配，这就是GFP_前缀的由来）是代表运行在内核空间的进程执行的。换句话说，这意味着调用它的函数正代表某个进程执行系统调用。使用GFP_KERNEL允许*kmalloc*在空闲内存较少时把当前进程转入休眠以等待一个页面。因此，使用GFP_KERNEL分配内存的函数必须是可重入的。在当前进程休眠时，内核会采取适当的行动，或者是把缓冲区的内容刷写到硬盘上，或者是从一个用户进程换出内存，以获取一个内存页面。

GFP_KERNEL分配标志并不是始终适用，有时*kmalloc*是在进程上下文之外被调用的，例如在中断处理例程、tasklet以及内核定时器中调用。这种情况下current进程就不应该休眠，驱动程序则应该换用GFP_ATOMIC标志。内核通常会为原子性的分配预留一些空闲页面。使用GFP_ATOMIC标志时，*kmalloc*甚至可以用掉最后一个空闲页面。不过如果连最后一页都没有了，分配就返回失败。

除了GFP_KERNEL和GFP_ATOMIC外，还有一些其他的标志可用于替换或补充这两个标志，不过这两个标志已经可以满足大多数驱动程序的需要了。所有的标志都定义在*<linux/gfp.h>*中，有个别的标志使用两个下划线作为前缀，比如__GFP_DMA。另外，还有一些符号表示这些标志的常用组合，它们没有这种前缀，并且有时称为“分配优先级”。这些符号包括：

GFP_ATOMIC
: 用于在中断处理例程或其他运行于进程上下文之外的代码中分配内存，不会休眠。

GFP_KERNEL
: 内核内存的通常分配方法，可能引起休眠。

GFP_USER
: 用于为用户空间页分配内存，可能会休眠。

GFP_HIGHUSER
: 类似于GFP_USER，不过如果有高端内存的话就从那里分配。我们在下一小节讨论高端内存相关的话题。

GFP_NOIO

GFP_NOFS
: 这两个标志的功能类似于GFP_KERNEL，但是为内核分配内存的工作方式添加了一些限制。具有GFP_NOFS标志的分配不允许执行任何文件系统调用，而GFP_NOIO

禁止任何I/O的初始化。这两个标志主要在文件系统和虚拟内存代码中使用，这些代码中的内存分配可休眠，但不应该发生递归的文件系统调用。

上面列出的分配标志可以和下面的标志“或”起来使用。下面这些标志控制如何进行分配：

__GFP_DMA
该标志请求分配发生在可进行DMA的内存区段中。具体的含义是平台相关的，我们将在下一小节中解释。

__GFP_HIGHMEM
这个标志表明要分配的内存可位于高端内存。

__GFP_COLD
通常，内存分配器会试图返回“缓存热（cache warm）”页面，即可在处理器缓存中找到的页面。相反，这个标志请求尚未使用的“冷”页面。对用于DMA读取的页面分配，可使用这个标志，因为这种情况下，页面存在于处理器缓存中没有多大帮助。有关DMA缓冲区分配的详细信息，可参阅第十五章的“直接内存访问”一节。

__GFP_NOWARN
该标志很少使用。它可以避免内核在无法满足分配请求时产生警告（使用*printk*）。

__GFP_HIGH
该标志标记了一个高优先级的请求，它允许为紧急状况而消耗由内核保留的最后一些页面。

__GFP_REPEAT
__GFP_NOFAIL
__GFP_NORETRY
上述标志告诉分配器在满足分配请求而遇到困难时应该采取何种行为。__GFP_REPEAT表示“努力再尝试一次”，它会重新尝试分配，但仍有可能失败。__GFP_NOFAIL标志告诉分配器始终不返回失败，它会努力满足分配请求。我们不鼓励使用__GFP_NOFAIL标志，因为在设备驱动程序中，从没有理由需要使用这个标志。最后，__GFP_NORETRY告诉分配器，如果所请求的内存不可获得，就立即返回。

内存区段

__GFP_DMA和__GFP_HIGHMEM的使用与平台相关，尽管在所有平台上都可以使用这两个标志。

Linux内核把内存分为三个区段：可用于DMA的内存、常规内存以及高端内存。通常的内存分配都发生在常规内存区，但通过设置上面介绍过的标志也可以请求在其他区段中分配。其思路是每种计算平台都必须知道如何把自己特定的内存范围归类到这三个区段中，而不是认为所有的RAM都一样。

可用于DMA的内存指存在于特别地址范围内的内存，外设可以利用这些内存执行DMA访问。在大多数健全的系统上，所有内存都位于这一区段。在x86平台上，DMA区段是RAM的前16 MB，老式的ISA设备可在该区段上执行DMA；PCI设备则无此限制。

高端内存是32位平台为了访问（相对）大量的内存而存在的一种机制。如果不首先完成一些特殊的映射，我们就无法从内核中直接访问这些内存，因此通常较难处理。但是，如果驱动程序要使用大量的内存，那么在能够使用高端内存的大系统上可以工作得更好。有关高端内存的工作方式及使用方法，可参阅第十五章“高端和低端内存”一节。

当一个新页面为满足*kmalloc*的要求被分配时，内核会创建一个内存区段的列表以供搜索。如果指定了__GFP_DMA标志，则只有DMA区段会被搜索：如果低地址段上没有可用内存，分配就会失败。如果没有指定特定的标志，则常规区段和DMA区段都会被搜索；而如果设置了__GFP_HIGHMEM标志，则所有三个区段都会被搜索以获取一个空闲页（然而要注意的是，*kmalloc*不能分配高端内存）。

在不均匀内存访问（nonuniform memory access，NUMA）系统上的情况更加复杂。作为通常的规则，分配器会试图在执行该分配的处理器的本地内存区中定位内存，当然存在一些方式来改变这种行为。

内存区段的背后机制在*mm/page_alloc.c*中实现，区段的初始化是平台相关的，通常在对应的*arch*树下的*mm/init.c*中。第十五章还会再次讨论这个问题。

size参数

内核负责管理系统物理内存，物理内存只能按页面进行分配。其结果是*kmalloc*和典型的用户空间的*malloc*在实现上有很大的差别。简单的基于堆的内存分配技术会遇到麻烦，因为页面边界的处理成为一个很棘手的问题。因此内核使用了特殊的基于页的分配技术，以最佳地利用系统RAM。

Linux处理内存分配的方法是，创建一系列的内存对象池，每个池中的内存块大小是固定一致的。处理分配请求时，就直接在包含有足够大的内存块的池中传递一个整块给请求者。内存管理机制相当复杂，其细节对设备驱动程序开发人员并不重要，所以就不仔细讨论了。

驱动程序开发人员应该记住一点，就是内核只能分配一些预定义的、固定大小的字节数组。如果申请任意数量的内存，那么得到的很可能会多一些，最多会到申请数量的两倍。另外，程序员应该记住，*kmalloc* 能处理的最小的内存块是 32 或者 64，到底是哪个则取决于当前体系结构使用的页面大小。

对 *kmalloc* 能够分配的内存块大小，存在一个上限。这个限制随着体系架构的不同以及内核配置选项的不同而变化。如果我们希望代码具有完整的可移植性，则不应该分配大于 128KB 的内存。但是，如果希望得到多于几千字节的内存，则最好使用除 *kmalloc* 之外的内存获取方法，我们将在本章稍后来描述这些方法。

后备高速缓存

设备驱动程序常常会反复地分配很多同一大小的内存块。既然内核已经维护了一组拥有同一大小内存块的内存池，那么为什么不为这些反复使用的块增加某些特殊的内存池呢？实际上，内核的确实现了这种形式的内存池，通常称为后备高速缓存（lookaside cache）。设备驱动程序通常不会涉及这种使用后备高速缓存的内存行为，但也有例外，Linux 2.6 中的 USB 和 SCSI 驱动程序就使用了这种高速缓存。

Linux 内核的高速缓存管理有时称为“slab 分配器”。因此，相关函数和类型在 *<linux/slab.h>* 中声明。slab 分配器实现的高速缓存具有 `kmem_cache_t` 类型，可通过调用 *kmem_cache_create* 创建：

```
kmem_cache_t *kmem_cache_create(const char *name, size_t size,
                                size_t offset,
                                unsigned long flags,
                                void (*constructor)(void *, kmem_cache_t *,
                                                    unsigned long flags),
                                void (*destructor)(void *, kmem_cache_t *,
                                                   unsigned long flags));
```

该函数创建一个新的高速缓存对象，其中可以容纳任意数目的内存区域，这些区域的大小都相同，由 `size` 参数指定。参数 `name` 与这个高速缓存相关联，其功能是保管一些信息以便追踪问题，它通常被设置为将要高速缓存的结构类型的名字。高速缓存保留指向该名称的指针，而不是复制其内容，因此，驱动程序应该将指向静态存储（通常可取直接字符串）的指针传递给这个函数。名称中不能包含空白。

`offset` 参数是页面中第一个对象的偏移量，它可以用来确保对已分配的对象进行某种特殊的对齐，但是最常用的就是 `0`，表示使用默认值。`flags` 控制如何完成分配，是一个位掩码，可取的值如下：

SLAB_NO_REAP

设置这个标志可以保护高速缓存在系统寻找内存的时候不会被减少。设置该标志通常不是好主意，因为我们不应该对内存分配器的自由做一些人为的、不必要的限制。

SLAB_HWCACHE_ALIGN

这个标志要求所有数据对象跟高速缓存行（cache line）对齐；实际的操作则依赖于主机平台的硬件高速缓存布局。如果在SMP机器上，高速缓存中包含有频繁访问的数据项的话，则该选项将是非常好的选择。但是，为了满足高速缓存行的对齐需求，必要的填白可能浪费大量内存。

SLAB_CACHE_DMA

这个标志要求每个数据对象都从可用于DMA的内存区段中分配。

还有一些标志可用于高速缓存分配的调试，详情请见 *mm/slab.c* 文件。但通常这些标志只在开发系统中通过内核配置选项而全局地设置。

`constructor` 和 `destructor` 参数是可选的函数（但是不能只有destructor而没有constructor）；前者用于初始化新分配的对象，而后者用于“清除”对象 —— 在内存空间被整个释放给系统之前。

constructor和destructor很有用，不过使用时有一些限制。constructor函数是在分配用于一组对象的内存时调用的。因为这些内存中可能会包含好几个对象，所以constructor函数可能会被多次调用。我们不能认为分配一个对象后随之就会调用一次constructor。类似地，destructor函数也有可能不是在一个对象释放后就立即被调用，而是在将来的某个未知的时间才被调用。constructor和destructor可能允许也可能不允许休眠，这要看是否向它们传递了SLAB_CTOR_ATOMIC标志（CTOR是 *constructor* 的简写）。

为了简便起见，程序员可以使用同一个函数同时作为constructor和destructor使用；当调用的是一个constructor函数的时候，slab分配器总是传递SLAB_CTOR_CONSTRUCTOR标志。

一旦某个对象的高速缓存被创建，就可以调用 *kmem_cache_alloc* 从中分配内存对象：

```
void *kmem_cache_alloc(kmem_cache_t *cache, int flags);
```

这里，参数 `cache` 是先前创建的高速缓存；参数flags和传递给 *kmalloc* 的相同，并且当需要分配更多内存来满足 *kmem_cache_alloc* 时，高速缓存还会利用这个参数。

释放一个内存对象时使用 *kmem_cache_free*：

```
void kmem_cache_free(kmem_cache_t *cache, const void *obj);
```

如果驱动程序代码中和高速缓存有关的部分已经处理完了(一个典型情况是模块被卸载的时候)，这时驱动程序应该释放它的高速缓存，如下所示:

```
int kmem_cache_destroy(kmem_cache_t *cache);
```

这个释放操作只有在已将从缓存中分配的所有对象都归还后才能成功。所以，模块应该检查*kmem_cache_destroy*的返回状态；如果失败，则表明模块中发生了内存泄漏(因为有一些对象被漏掉了)。

使用后备式缓存带来的另一个好处是内核可以统计高速缓存的使用情况。高速缓存的使用统计情况可以从*/proc/slabinfo*获得。

基于slab高速缓存的scull：scullc

现在该举个例子了。*scullc*是*scull*模块的一个缩减版本，只实现了裸设备——即持久的内存区。与*scull*使用*kmalloc*不同的是，*scullc*使用内存高速缓存。数据对象的大小可以在编译或加载时修改，但不能在运行时修改——那样需要创建一个新的内存高速缓存，而这里不必处理那些不需要的细节问题。

*scullc*是一个完整的例子，可以用于测试slab分配器。它和*scull*只有几行代码的不同。首先，我们必须声明自己的slab高速缓存:

```
/* 声明一个高速缓存指针，它将用于所有设备 */
kmem_cache_t *scullc_cache;
```

slab高速缓存的创建代码如下所示(在模块装载阶段):

```
/* scullc_init: 为我们的量子创建一个高速缓存 */
scullc_cache = kmem_cache_create("scullc", scullc_quantum,
        0, SLAB_HWCACHE_ALIGN, NULL, NULL); /* 没有ctor/dtor */
if (!scullc_cache) {
    scullc_cleanup();
    return -ENOMEM;
}
```

下面是分配内存量子的代码:

```
/* 使用内存高速缓存来分配一个量子 */
if (!dptr->data[s_pos]) {
    dptr->data[s_pos] = kmem_cache_alloc(scullc_cache, GFP_KERNEL);
    if (!dptr->data[s_pos])
        goto nomem;
    memset(dptr->data[s_pos], 0, scullc_quantum);
}
```

下面的代码将释放内存:

```
for (i = 0; i < qset; i++)
if (dptr->data[i])
        kmem_cache_free(scullc_cache, dptr->data[i]);
```

最后，在模块卸载期间，我们必须将高速缓存返回给系统：

```
/* scullc_cleanup: 释放量子使用的高速缓存 */
if (scullc_cache)
    kmem_cache_destroy(scullc_cache);
```

和*scull*相比，*scullc*的最主要差别是运行速度略有提高，并且对内存的利用率更佳。由于数据对象是从内存池中分配的，而内存池中的所有内存块都具有同样大小，所以这些数据对象在内存中的位置排列达到了最大程度的密集，相反的，*scull*的数据对象则会引入不可预测的内存碎片。

内存池

内核中有些地方的内存分配是不允许失败的。为了确保这种情况下的成功分配，内核开发者建立了一种称为内存池（或者“mempool”）的抽象。内存池其实就是某种形式的后备高速缓存，它试图始终保存空闲的内存，以便在紧急状态下使用。

内存池对象的类型为mempool_t（在*<linux/mempool.h>*中定义），可使用*mempool_create*来建立内存池对象：

```
mempool_t *mempool_create(int min_nr,
                          mempool_alloc_t *alloc_fn,
                          mempool_free_t *free_fn,
                          void *pool_data);
```

min_nr参数表示的是内存池应始终保持的已分配对象的最少数目。对象的实际分配和释放由alloc_fn和free_fn函数处理，其原型如下：

```
typedef void *(mempool_alloc_t)(int gfp_mask, void *pool_data);
typedef void (mempool_free_t)(void *element, void *pool_data);
```

*mempool_create*的最后一个参数，即pool_data，被传入alloc_fn和free_fn。

如有必要，我们可以为mempool编写特定用途的函数来处理内存分配。但是，通常我们仅会让内核的slab分配器为我们处理这个任务。内核中有两个函数（*mempool_alloc_slab*和*mempool_free_slab*），它们的原型和上述内存池分配原型匹配，并利用*kmem_cache_alloc*和*kmem_cache_free*处理内存分配和释放。因此，构造内存池的代码通常如下所示：

```
cache = kmem_cache_create(. . .);
pool = mempool_create(MY_POOL_MINIMUM,
```

```
                    mempool_alloc_slab, mempool_free_slab,
                    cache);
```

在建立内存池之后，可如下所示分配和释放对象：

```
void *mempool_alloc(mempool_t *pool, int gfp_mask);
void mempool_free(void *element, mempool_t *pool);
```

在创建 mempool 时，就会多次调用分配函数为预先分配的对象创建内存池。之后，对 *mempool_alloc* 的调用将首先通过分配函数获得该对象；如果该分配失败，就会返回预先分配的对象（如果存在的话）。如果使用 *mempool_free* 释放一个对象，则如果预先分配的对象数目小于要求的最低数目，就会将该对象保留在内存池中；否则，该对象会返回给系统。

我们可以利用下面的函数来调整 mempool 的大小：

```
int mempool_resize(mempool_t *pool, int new_min_nr, int gfp_mask);
```

如果对该函数的调用成功，将把内存池的大小调整为至少有 *new_min_nr* 个预分配对象。

如果不再需要内存池，可使用下面的函数将其返回给系统：

```
void mempool_destroy(mempool_t *pool);
```

在销毁mempool之前，必须将所有已分配的对象返回到内存池中，否则会导致内核oops。

如果读者计划在自己的驱动程序中使用mempool，则应记住下面这点：mempool会分配一些内存块，空闲且不会真正得到使用。因此，使用mempool很容易浪费大量内存。几乎在所有情况下，最好不使用mempool而是处理可能的分配失败。如果驱动程序存在某种方式可以响应分配的失败，而不会导致对系统一致性的破坏，则应该使用这种方式，也就是说，应尽量避免在驱动程序代码中使用 mempool。

get_free_page 和相关函数

如果模块需要分配大块的内存，使用面向页的分配技术会更好些。整页的分配还有其他优点，以后会在第十五章介绍。

分配页面可使用下面的函数：

`get_zeroed_page(unsigned int flags);`

返回指向新页面的指针并将页面清零。

`__get_free_page(unsigned int flags);`

类似于 *get_zeroed_page*，但不清零页面。

__get_free_pages(unsigned int flags, unsigned int order);

分配若干（物理连续的）页面，并返回指向该内存区域第一个字节的指针，但不清零页面。

参数flags的作用和*kmalloc*中的一样；通常使用GFP_KERNEL或GFP_ATOMIC，也许还会加上__GFP_DMA标志（申请可用于ISA直接内存访问操作的内存）或者__GFP_HIGHMEM标志（使用高端内存）（注2）。参数order是要申请或释放的页面数的以2为底的对数（即$\log_2 N$）。例如，order（阶数）为0表示一个页面，order为3表示8个页面。如果order太大，而又没有那么大的连续区域可以分配，就会返回失败。*get_order*函数使用一个整数参数，可根据宿主平台上的大小（必须是2的幂）返回order值。可允许的最大order值是10或者11（对应于1024或2048个页），这依赖于体系结构。但是，相比具有大量内存的刚刚启动的系统而言，以阶数值为10进行分配而成功的机会很小。

如果读者对此好奇，*/proc/buddyinfo*可告诉你系统中每个内存区段上每个阶数下可获得的数据块数目。

当程序不再需要使用页面时，它可以使用下列函数之一来释放它们。第一个函数是一个宏，展开后就是对第二个函数的调用：

```
void free_page(unsigned long addr);
void free_pages(unsigned long addr, unsigned long order);
```

如果试图释放和先前分配数目不等的页面，内存映射关系就会被破坏，随后系统就会出错。

值得强调的是，只要符合和*kmalloc*同样的规则，*get_free_pages*和其他函数可以在任何时间调用。某些情况下函数分配内存时会失败，特别是在使用了GFP_ATOMIC的时候。因此，调用了这些函数的程序在分配出错时都应提供相应的处理。

尽管kmalloc(GFP_KERNEL)在没有空闲内存时有时会失败，但内核总会尽可能满足这个内存分配请求。因此，如果分配太多内存，系统的响应性能就很容易降下来。例如，如果往*scull*设备写入大量数据，计算机可能就会死掉；当系统为满足*kmalloc*分配请求而试图换出尽可能多的内存页时，就会变得很慢。所有资源都被贪婪的设备所吞噬，计算机很快就变的无法使用了；此时甚至已经无法为解决这个问题而生成新的进程。我们没有在*scull*模块中提到这个问题，因为它只是个例子，并不会真正在多用户系统中使

注2：尽管*alloc_pages*（稍后讲述）实际应该用于分配高端内存页，但出于某些原因的考虑，我们将在第十五章对其进行讲述。

用。但作为一个编程者必须要小心，因为模块是特权代码，会带来新的系统安全漏洞，例如很可能会造成 DoS（denail-of-service，拒绝服务攻击）安全漏洞。

使用整页的 scull: scullp

为了实际测试页面分配，我们编写了 *scullp* 模块。就像前面介绍的 *scullc* 一样，它是一个缩减了的 *scull*。

scullp 分配的内存数量是一个或数个整页：`scullp_order` 变量默认为 0，但可以在编译或加载时更改。

下列代码说明了它如何分配内存：

```
/* 下面分配单个量子 */
if (!dptr->data[s_pos]) {
    dptr->data[s_pos] =
        (void *)_ _get_free_pages(GFP_KERNEL, dptr->order);
    if (!dptr->data[s_pos])
        goto nomem;
    memset(dptr->data[s_pos], 0, PAGE_SIZE << dptr->order);
}
```

scullp 中释放内存的代码如下：

```
/* 这段代码释放整个量子集 */
for (i = 0; i < qset; i++)
    if (dptr->data[i])
        free_pages((unsigned long)(dptr->data[i]),
                dptr->order);
```

从用户的角度看，可以感觉到的差别就是速度快了一些，并且内存利用率更高了，因为不会有内部的内存碎片。我们运行了测试程序，把 4MB 的数据从 *scull0* 拷贝到 *scull1*，然后再从 *scullp0* 拷贝到 *scullp1*；结果表明处理器在内核空间的使用率有所提高。

但性能的提高并不多，因为 *kmalloc* 已经运行得很快。基于页的分配策略的优点实际不在速度上，而在于更有效地使用了内存。按页分配不会浪费内存空间，而用 *kmalloc* 函数则会因分配粒度的原因而浪费一定数量的内存。

但是使用 *__get_free_page* 函数的最大优点是这些分配的页面完全属于我们自己，而且在理论上可以通过适当地调整页表将它们合并成一个线性区域。例如，可以允许用户进程对这些单一但互不相关的页面分配得到的内存区域进行 *mmap*。我们将在第十五章中讨论这种操作，届时将演示 *scullp* 如何提供内存映射，而 *scull* 无法提供这种操作。

alloc_pages 接口

为完整起见，本节将介绍内存分配的另一个接口，但在第十五章才会使用这个接口。现在，我们只要知道struct page是内核用来描述单个内存页的数据结构就足够了。我们将看到，内核中有许多地方需要使用page结构，尤其在需要使用高端内存（高端内存在内核空间没有对应不变的地址）的地方。

Linux 页分配器的核心代码是称为 *alloc_pages_node* 的函数：

```
struct page *alloc_pages_node(int nid, unsigned int flags,
                              unsigned int order);
```

这个函数具有两个变种（它们只是简单的宏），大多数情况下我们使用这两个宏：

```
struct page *alloc_pages(unsigned int flags, unsigned int order);
struct page *alloc_page(unsigned int flags);
```

核心函数 *alloc_pages_node* 要求传入三个参数。nid是NUMA节点的ID号（注3），表示要在其中分配内存，flags是通常的GFP_分配标志，而order是要分配的内存大小。该函数的返回值是指向第一个page结构（可能返回多个页）的指针，它描述了已分配的内存；或者在失败时返回NULL。

alloc_pages通过在当前的NUMA节点上分配内存而简化了 *alloc_pages_node* 函数，它将 *numa_node_id* 的返回值作为nid参数而调用了 *alloc_pages_node* 函数。另外，*alloc_page* 函数显然忽略了order参数而只分配单个页面。

为了释放通过上述途径分配的页面，我们应使用下面的函数：

```
void _ _free_page(struct page *page);
void _ _free_pages(struct page *page, unsigned int order);
void free_hot_page(struct page *page);
void free_cold_page(struct page *page);
```

如果读者知道某个页面中的内容是否驻留在处理器高速缓存中，则应该使用 *free_hot_page*（用于驻留在高速缓存中的页）或者 *free_cold_page* 和内核通信。这个信息可帮助内存分配器优化内存的使用。

注3：NUMA计算机是多处理器系统，其中的内存对特定处理器组（节点）来讲是“本地的”。访问本地内存要比访问非本地内存快。在这类系统中，在正确节点上的内存分配非常重要。但是驱动程序作者通常不用担心NUMA问题。

vmalloc及其辅助函数

下面要介绍的内存分配函数是*vmalloc*，它分配虚拟地址空间的连续区域。尽管这段区域在物理上可能是不连续的（要访问其中的每个页面都必须独立地调用函数*alloc_page*），内核却认为它们在地址上是连续的。*vmalloc*在发生错误时返回0（NULL地址），成功时返回一个指针，该指针指向一个线性的、大小最少为size的线性内存区域。

我们在这里描述*vmalloc*的原因是，它是Linux内存分配机制的基础。但是，我们要注意在大多数情况下不鼓励使用*vmalloc*。通过*vmalloc*获得的内存使用起来效率不高，而且在某些体系架构上，用于*vmalloc*的地址空间总量相对较小。如果希望将使用*vmalloc*的代码提交给内核主线代码，则可能会受到冷遇。如果可能，应该直接和单个的页面打交道，而不是使用*vmalloc*。

虽然这么说，但我们还是要看看如何使用*vmalloc*。该函数的原型及其相关函数（*ioremap*，并不是严格的分配函数，将在本节后面讨论）如下所示：

```
#include <linux/vmalloc.h>

void *vmalloc(unsigned long size);
void vfree(void * addr);
void *ioremap(unsigned long offset, unsigned long size);
void iounmap(void * addr);
```

要强调的是，由*kmalloc*和*__get_free_pages*返回的内存地址也是虚拟地址，其实际值仍然要由MMU（内存管理单元，通常是CPU的组成部分）处理才能转为物理内存地址（注4）。*vmalloc*在如何使用硬件上没有区别，区别在于内核如何执行分配任务上。

*kmalloc*和*__get_free_pages*使用的（虚拟）地址范围与物理内存是一一对应的，可能会有基于常量PAGE_OFFSET的一个偏移。这两个函数不需要为该地址段修改页表。但另一方面，*vmalloc*和*ioremap*使用的地址范围完全是虚拟的，每次分配都要通过对页表的适当设置来建立（虚拟）内存区域。

可以通过比较内存分配函数返回的指针来发现这种差别。在某些平台上（如x86），*vmalloc*返回的地址仅仅比*kmalloc*返回的地址高一些；而在其他平台上（如MIPS和IA-

注4：实际上，某些体系架构定义了保留的“虚拟”地址范围，用于寻址物理内存。遇到这种情况时，Linux内核会利用这种特性，内核和*__get_free_pages*地址均位于这种内存范围。其中的区别对设备驱动程序是透明的，对不直接涉及内存管理子系统的其他内核代码来说也是透明的。

64)，它们就完全属于不同的地址范围了。*vmalloc* 可以获得的地址在VMALLOC_START到 VMALLOC_END 的范围中。这两个符号都在 *<asm/pgtable.h>* 中定义。

用 *vmalloc* 分配得到的地址是不能在微处理器之外使用的，因为它们只在处理器的内存管理单元上才有意义。当驱动程序需要真正的物理地址时（像外设用以驱动系统总线的DMA 地址），就不能使用 *vmalloc* 了。使用 *vmalloc* 函数的正确场合是在分配一大块连续的、只在软件中存在的、用于缓冲的内存区域的时候。注意 *vmalloc* 的开销要比 *__get_free_pages* 大，因为它不但获取内存，还要建立页表。因此，用 *vmalloc* 函数分配仅仅一页的内存空间是不值得的。

使用 *vmalloc* 函数的一个例子函数是 *create_module* 系统调用，它利用 *vmalloc* 函数来获取装载模块所需的内存空间。在调用 *insmod* 来重定位模块代码后，接着会调用 *copy_from_user* 函数把模块代码和数据复制到分配而得的空间内。这样，模块看来像是在连续的内存空间内。但通过检查 */proc/ksyms* 文件就能发现模块导出的内核符号和内核本身导出的符号分布在不同的内存范围上。

用 *vmalloc* 分配得到的内存空间要用 *vfree* 函数来释放，这就像要用 *kfree* 函数来释放 *kmalloc* 函数分配得到的内存空间一样。

和 *vmalloc* 一样，*ioremap* 也建立新的页表，但和 *vmalloc* 不同的是，*ioremap* 并不实际分配内存。*ioremap* 的返回值是一个特殊的虚拟地址，可以用来访问指定的物理内存区域，这个虚拟地址最后要调用 *iounmap* 来释放掉。

ioremap 更多用于映射（物理的）PCI缓冲区地址到（虚拟的）内核空间。例如，可以用来访问PCI视频设备的帧缓冲区；该缓冲区通常被映射到高物理地址，超出了系统初始化时建立的页表地址范围。PCI的详细内容将在第十二章中讨论。

要注意，为了保持可移植性，不应把 *ioremap* 返回的地址当作指向内存的指针而直接访问。相反，应该使用 *readb* 或其他 I/O 函数（在第九章“使用 I/O 内存”一节中介绍）。这是因为，在如 Alpha 的一些平台上，由于 PCI 规范和 Alpha 处理器在数据传输方式上的差异，不能直接把 PCI 内存区映射到处理器的地址空间。

ioremap 和 *vmalloc* 函数都是面向页的（它们都会修改页表），因此重新定位或分配的内存空间实际上都会上调到最近的一个页边界。*ioremap* 通过把重新映射的地址向下下调到页边界，并返回在第一个重新映射页面中的偏移量的方法模拟了不对齐的映射。

vmalloc 函数的一个小缺点是它不能在原子上下文中使用，因为它的内部实现调用了 *kmalloc(GFP_KERNEL)* 来获取页表的存储空间，因而可能休眠。但这不是什么问题——如果 *__get_free_page* 函数都还不能满足中断处理例程的需求的话，那应该修改软件的设计了。

使用虚拟地址的 scull: scullv

scullv 模块使用了 *vmalloc*。和 *scullp* 一样，这个模块也是 *scull* 的一个缩减版本，只是使用了不同的分配函数来获取设备用于储存数据的内存空间。

该模块每次分配16页的内存。这里的内存分配使用了较大的数据块以获取比 *scullp* 更好的性能，并且展示了为什么使用其他分配技术会更耗时。用 *__get_free_pages* 函数来分配一页以上的内存空间容易出错，而且即使成功了也比较慢。在前面我们已经看到，用 *vmalloc* 分配几个页时比其他函数要快一些，但由于存在建立页表的开销，所以当只分配一页时却会慢一些。*scullv* 设计得和 *scullp* 很相似。`order` 参数指定每次要分配的内存空间的“阶数”，默认为4。*scullv* 和 *scullp* 的唯一差别是在分配管理上。下面的代码用 *vmalloc* 获取新内存：

```
/* 使用虚拟地址分配一个量子 */
if (!dptr->data[s_pos]) {
    dptr->data[s_pos] =
        (void *)vmalloc(PAGE_SIZE << dptr->order);
    if (!dptr->data[s_pos])
        goto nomem;
    memset(dptr->data[s_pos], 0, PAGE_SIZE << dptr->order);
}
```

这段代码释放内存：

```
/* 释放量子集 */
for (i = 0; i < qset; i++)
    if (dptr->data[i])
        vfree(dptr->data[i]);
```

如果在编译这两个模块时都启用了调试，就可以通过读取它们在 */proc* 下创建的文件来查看数据分配过程。下面的快照取自一个 x86_64 系统：

```
salma% cat /tmp/bigfile > /dev/scullp0; head -5 /proc/scullpmem
Device 0: qset 500, order 0, sz 1535135
  item at 000001001847da58, qset at 000001001db4c000
       0:1001db56000
       1:1003d1c7000

salma% cat /tmp/bigfile > /dev/scullv0; head -5 /proc/scullvmem

Device 0: qset 500, order 4, sz 1535135
  item at 000001001847da58, qset at 0000010013dea000
       0:ffffff0001177000
       1:ffffff0001188000
The following output, instead, came from an x86 system:
rudo% cat /tmp/bigfile > /dev/scullp0; head -5 /proc/scullpmem

Device 0: qset 500, order 0, sz 1535135
  item at ccf80e00, qset at cf7b9800
```

```
        0:ccc58000
        1:cccdd000

rudo%  cat /tmp/bigfile > /dev/scullv0; head -5 /proc/scullvmem

Device 0: qset 500, order 4, sz 1535135
  item at cfab4800, qset at cf8e4000
        0:d087a000
        1:d08d2000
```

这些数值说明了二者行为上的差别。在x86_64平台上，物理地址和虚拟地址被映射到完全不同的地址范围（0x100和0xffffff00），而在x86平台上，*vmalloc*返回的虚拟地址就在用于映射物理内存的地址之上。

per-CPU变量

per-CPU（每CPU）变量是2.6内核的一个有趣特性。当建立一个per-CPU变量时，系统中的每个处理器都会拥有该变量的特有副本。这看起来有些奇怪，但它有其优点。对per-CPU变量的访问（几乎）不需要锁定，因为每个处理器在其自己的副本上工作。per-CPU变量还可以保存在对应处理器的高速缓存中，这样，就可以在频繁更新时获得更好的性能。

关于per-CPU变量使用的例子可见于网络子系统中。内核维护着大量计数器，这些计数器跟踪已接收到的各类数据包数量，而这些计数器每秒可能被更新上千次。网络子系统的开发者将这些统计用的计数器放在了per-CPU变量中，这样，他们就不需要处理缓存和锁定问题，而更新可在不用锁的情况下快速完成。在用户空间偶尔请求这些计数器的值时，只需将每个处理器的版本相加并返回合计值即可。

用于per-CPU变量的声明可见于*<linux/percpu.h>*中。要在编译期间创建一个per-CPU变量，可使用下面的宏：

```
DEFINE_PER_CPU(type, name);
```

如果该变量（称为name）是一个数组，需在type中包含数组的维数。这样，具有三个整数的per-CPU数组变量可通过下面的语句建立：

```
DEFINE_PER_CPU(int[3], my_percpu_array);
```

对per-CPU变量的操作几乎不使用任何锁定即可完成。但要记得2.6内核是抢占式的；也就是说，当处理器在修改某个per-CPU变量的临界区中间，可能会被抢占，因此应该避免这种情况的发生。我们还应该避免进程正在访问一个per-CPU变量时被切换到另一个处理器上运行。为此，我们应该显式地调用*get_cpu_var*宏访问某给定变量的当前处

理器副本，结束后调用*put_cpu_var*。对*get_cpu_var*的调用将返回当前处理器变量版本的lvalue值，并禁止抢占。因为返回的是lvalue，因此可直接赋值或者操作。例如，网络代码对一个计数器的递增使用了下面的两条语句：

```
get_cpu_var(sockets_in_use)++;
put_cpu_var(sockets_in_use);
```

我们可以使用下面的宏访问其他处理器的变量副本：

```
per_cpu(variable, int cpu_id);
```

如果我们要编写的代码涉及到多个处理器的per-CPU变量，这时则需要采用某种锁定机制来确保访问安全。

动态分配per-CPU变量也是可能的。这时，应使用下面的函数分配变量：

```
void *alloc_percpu(type);
void *_ _alloc_percpu(size_t size, size_t align);
```

在大多数情况下可使用*alloc_percpu*完成分配工作；但如果需要特定的对齐，则应该调用*__alloc_percpu*函数。不管使用哪个函数，可使用*free_percpu*将per-CPU变量返回给系统。对动态分配的per-CPU变量的访问通过*per_cpu_ptr*完成：

```
per_cpu_ptr(void *per_cpu_var, int cpu_id);
```

这个宏返回指向对应于给定cpu_id的per_cpu_var版本的指针。如果打算读取该变量的其他CPU版本，则可以引用该指针并进行相关操作。但是，如果正在操作当前处理器的版本，则应该首先确保自己不会被切换到其他处理器上运行。如果对per-CPU变量的整个访问发生在拥有某个自旋锁的情况下，则不会出现任何问题。但是，在使用该变量的时候通常需要使用*get_cpu*来阻塞抢占。这样，使用动态per-CPU变量的代码类似下面所示：

```
int cpu;

cpu = get_cpu()
ptr = per_cpu_ptr(per_cpu_var, cpu);
/* 使用 ptr */
put_cpu();
```

如果使用编译期间的per-CPU变量，则*get_cpu_var*和*put_cpu_var*宏将处理这些细节。但动态的per-CPU变量需要更明确的保护。

Per-CPU变量可以导出给模块，但是必须使用上述宏的特殊版本：

```
EXPORT_PER_CPU_SYMBOL(per_cpu_var);
EXPORT_PER_CPU_SYMBOL_GPL(per_cpu_var);
```

要在模块中访问这样一个变量，则应将其声明如下：

```
DECLARE_PER_CPU(type, name);
```

使用*DECLARE_PER_CPU*（而不是*DEFINE_PER_CPU*），将告诉编译器要使用一个外部引用。

如果读者打算使用per-CPU变量来建立简单的整数计数器，可参考*<linux/percpu_counter.h>*中已封装好的实现。最后要注意，在某些体系架构上，per-CPU变量可使用的地址空间是受限制的。因此，如果要创建per-CPU变量，则应该保持这些变量较小。

获取大的缓冲区

我们在前面的小节中提到，大的、连续内存缓冲区的分配易流于失败。系统内存会随着时间的流逝而碎片化，这导致无法获得真正的大内存区域。因为，不需要大的缓冲区也可以有其他途径来完成自己的工作，因此内核开发者并没有将大缓冲区分配工作作为高优先级的任务来计划。在试图获得大内存区之前，我们应该仔细考虑其他的实现途径。到目前为止，执行大的I/O操作的最好方式是通过离散/聚集操作，我们将在第十章的“离散/聚集映射”中讨论这种操作。

在引导时获得专用缓冲区

如果的确需要连续的大块内存用作缓冲区，就最好在系统引导期间通过请求内存来分配。在引导时就进行分配是获得大量连续内存页面的唯一方法，它绕过了*__get_free_pages*函数在缓冲区大小上的最大尺寸和固定粒度的双重限制。在引导时分配缓冲区有点“脏”，因为它通过保留私有内存池而跳过了内核的内存管理策略。这种技术比较粗暴也很不灵活，但也是最不容易失败的。显然，模块不能在引导时分配内存，而只有直接链接到内核的设备驱动程序才能在引导时分配内存。

还有一个值得注意的问题是，对于普通用户来说引导时的分配不是一个切实可用的选项，因为这种机制只对链接到内核映像中的代码可用。要安装或替换使用了这种分配技术的驱动程序，就只能重新编译内核并重启计算机。

内核被引导时，它可以访问系统所有的物理内存，然后调用各个子系统的初始化函数进行初始化，它允许初始化代码分配私有的缓冲区，同时减少了留给常规系统操作的RAM数量。

通过调用下列函数之一则可完成引导时的内存分配：

```
#include <linux/bootmem.h>
void *alloc_bootmem(unsigned long size);
void *alloc_bootmem_low(unsigned long size);
void *alloc_bootmem_pages(unsigned long size);
void *alloc_bootmem_low_pages(unsigned long size);
```

这些函数要么分配整个页（若以 _pages 结尾），要么分配不在页面边界上对齐的内存区。除非使用具有 _low 后缀的版本，否则分配的内存可能会是高端内存。如果我们正在为设备驱动程序分配缓冲区，则可能希望将其用于 DMA 操作，而高端内存并不总是支持 DMA 操作；这样，我们可能需要使用上述函数的一个 _low 变种。

很少会释放引导时分配的内存，而且也没有任何办法可将这些内存再次拿回。但是，内核也提供了一种释放这种内存的接口：

```
void free_bootmem(unsigned long addr, unsigned long size);
```

注意，通过上述函数释放的部分页面不会返回给系统 —— 但是，如果我们使用这种技术，则其实已经分配得到了一些完整的页面。

如果必须使用引导时的分配，则应该将驱动程序直接链接到内核。关于直接链接到内核的实现细节，可参阅内核源代码中 *Documentation/kbuild* 目录下的文件。

快速参考

与内存分配有关的函数和符号如下：

```
#include <linux/slab.h>
void *kmalloc(size_t size, int flags);
void kfree(void *obj);
```

最常用的内存分配接口。

```
#include <linux/mm.h>
GFP_USER
GFP_KERNEL
GFP_NOFS
GFP_NOIO
GFP_ATOMIC
```

用来控制内存分配执行方式的标志，其排列从最少限制到最多限制。GFP_USER和GFP_KERNEL优先级允许当前进程休眠以满足分配请求。GFP_NOFS和GFP_NOIO分别禁用文件系统操作和所有的I/O操作，而GFP_ATOMIC根本不允许休眠。

__GFP_DMA

__GFP_HIGHMEM

__GFP_COLD

__GFP_NOWARN

__GFP_HIGH

__GFP_REPEAT

__GFP_NOFAIL

__GFP_NORETRY

上述标志在分配内存时修改内核的行为。

```
#include <linux/malloc.h>
kmem_cache_t *kmem_cache_create(char *name, size_t size, size_t offset,
  unsigned long flags, constructor(), destructor());
int kmem_cache_destroy(kmem_cache_t *cache);
```

创建和销毁一个包含固定大小内存块的slab高速缓存，我们可以从这个高速缓存中分配具有固定大小的对象。

SLAB_NO_REAP

SLAB_HWCACHE_ALIGN

SLAB_CACHE_DMA

在创建高速缓存时指定的标志。

SLAB_CTOR_ATOMIC

SLAB_CTOR_CONSTRUCTOR

可由分配器传递给 constructor 和 destructor 函数的标志。

```
void *kmem_cache_alloc(kmem_cache_t *cache, int flags);
void kmem_cache_free(kmem_cache_t *cache, const void *obj);
```

从高速缓存中分配和释放一个对象。

/proc/slabinfo

包含有 slab 高速缓存使用统计信息的虚拟文件。

```
#include <linux/mempool.h>
mempool_t *mempool_create(int min_nr, mempool_alloc_t *alloc_fn,
  mempool_free_t *free_fn, void *data);
void mempool_destroy(mempool_t *pool);
```

用于创建内存池的函数。内存池通过保留预分配项的“急用链表”来避免内存分配的失败。

```
void *mempool_alloc(mempool_t *pool, int gfp_mask);
void mempool_free(void *element, mempool_t *pool);
```

从内存池分配或者释放对象的函数。

```
unsigned long get_zeroed_page(int flags);
unsigned long _ _get_free_page(int flags);
unsigned long _ _get_free_pages(int flags, unsigned long order);
```

面向页的分配函数。*get_zeroed_page* 返回单个已清零的页面，而其他所有调用不进行页面的初始化。

```
int get_order(unsigned long size);
```

根据 PAGE_SIZE 返回当前平台上和 *size* 关联的分配阶数。该函数的参数必须是 2 的幂，而返回值至少为 0。

```
void free_page(unsigned long addr);
void free_pages(unsigned long addr, unsigned long order);
```

这些函数释放面向页分配的内存。

```
struct page *alloc_pages_node(int nid, unsigned int flags, unsigned int order);
struct page *alloc_pages(unsigned int flags, unsigned int order);
struct page *alloc_page(unsigned int flags);
```

Linux 内核中最底层页分配器的所有变种。

```
void __free_page(struct page *page);
void __free_pages(struct page *page, unsigned int order);
void free_hot_page(struct page *page);
void free_cold_page(struct page *page);
```

用于释放由 *alloc_page* 的某种形式分配的页的不同函数。

```
#include <linux/vmalloc.h>
void * vmalloc(unsigned long size);
void vfree(void * addr);
#include <asm/io.h>
void * ioremap(unsigned long offset, unsigned long size);
void iounmap(void *addr);
```

这些函数分配或释放连续的虚拟地址空间。*ioremap* 通过虚拟地址访问物理内存，而 *vmalloc* 分配空闲页面。使用 *ioremap* 映射的区域用 *iounmap* 释放，而从 *vmalloc* 获得的页面用 *vfree* 释放。

```
#include <linux/percpu.h>
DEFINE_PER_CPU(type, name);
DECLARE_PER_CPU(type, name);
```

定义和声明per-CPU变量的宏。

```
per_cpu(variable, int cpu_id)
get_cpu_var(variable)
put_cpu_var(variable)
```

用于访问静态声明的per-CPU变量的宏。

```
void *alloc_percpu(type);
void *__alloc_percpu(size_t size, size_t align);
void free_percpu(void *variable);
```

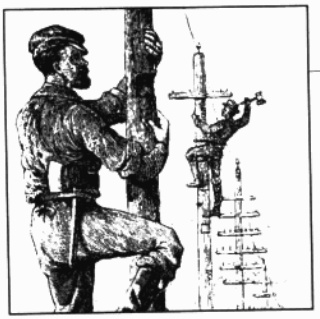
执行per-CPU变量的运行时分配和释放的函数。

```
int get_cpu();
void put_cpu();
per_cpu_ptr(void *variable, int cpu_id)
```

*get_cpu*获得对当前处理器的引用（因此避免抢占以及切换到其他处理器）并返回处理器的ID号；而*put_cpu*返回该引用。为了访问动态分配的per-CPU变量，应使用*per_cpu_ptr*，并传递要访问的变量版本的CPU ID号。对某个动态的per-CPU变量的当前CPU版本的操作，应该包含在对*get_cpu*和*put_cpu*的调用中间。

```
#include <linux/bootmem.h>
void *alloc_bootmem(unsigned long size);
void *alloc_bootmem_low(unsigned long size);
void *alloc_bootmem_pages(unsigned long size);
void *alloc_bootmem_low_pages(unsigned long size);
void free_bootmem(unsigned long addr, unsigned long size);
```

在系统引导期间执行内核分配和释放的函数。这些函数只能在直接链接到内核的驱动程序中使用。



第九章
与硬件通信

尽管摆弄*scull*以及其他一些玩具程序对理解Linux设备驱动程序的软件接口很有帮助，但实现真正的设备仍要涉及实际的硬件。设备驱动程序是软件概念和硬件电路之间的一个抽象层，因此，两方面都要讨论。到现在为止，我们已经详细讨论了软件概念上的一些细节，而本章将讨论另一方面，介绍驱动程序在Linux平台之上如何在保持可移植性的前提下访问I/O端口和I/O内存。

和前面一样，本章尽可能不针对特定的硬件设备。但是在需要示例的场合，我们将使用简单的数字I/O端口（比如标准的PC并口）来讲解I/O指令，并使用普通的帧缓冲区显存来讲解内存映射I/O。

我们选择简单的数字I/O是因为它是最简单的输入/输出端口。几乎所有的计算机上都有并口，它实现了裸的I/O：写到设备的数据位出现在输出引脚上，而输入引脚的电压值可以由处理器直接获取。实践中，我们必须将LED或者打印机连接到并口上才能真正“看到”数字I/O操作的结果，但底层硬件非常容易使用。

I/O端口和I/O内存

每种外设都通过读写寄存器进行控制。大部分外设都有几个寄存器，不管是在内存地址空间还是在I/O地址空间，这些寄存器的访问地址都是连续的。

在硬件层，内存区域和I/O区域没有概念上的区别：它们都通过向地址总线和控制总线发送电平信号（比如读和写信号）（注1）进行访问，再通过数据总线读写数据。

注1：并非所有计算机平台都使用读和写信号；有些使用不同的方式访问外部电路。不过这些区别对软件是透明的，为简化讨论，这里假定所有平台都用读和写信号。

一些CPU制造厂商在它们的芯片中使用单一地址空间，而另一些则为外设保留了独立的地址空间，以便和内存区分开来。一些处理器（主要是x86家族的）还为I/O端口的读和写提供了独立的线路，并且使用特殊的CPU指令访问端口。

因为外设要与外围总线相匹配，而最流行的I/O总线是基于个人计算机模型的，所以即使原本没有独立的I/O端口地址空间的处理器，在访问外设时也要模拟成读写I/O端口，这通常由外部芯片组或CPU核心中的附加电路来实现。后一种方式只在嵌入式的微处理器中比较多见。

基于同样的原因，Linux在所有的计算机平台上都实现了I/O端口，包括使用单一地址空间的CPU在内。端口操作的具体实现有时依赖于宿主计算机的特定型号和构造（因为不同的型号使用不同的芯片组把总线操作映射到内存地址空间）。

即使外设总线为I/O端口保留了分离的地址空间，也不是所有的设备都会把寄存器映射到I/O端口。ISA设备普遍使用I/O端口，而大多数PCI设备则把寄存器映射到某个内存地址区段。这种I/O内存通常是首选方案，因为不需要特殊的处理器指令；而且CPU核心访问内存更有效率，同时在访问内存时，编译器在寄存器分配和寻址方式的选择上也有更多的自由。

I/O寄存器和常规内存

尽管硬件寄存器和内存非常相似，但程序员在访问I/O寄存器的时候必须注意避免由于CPU或编译器不恰当的优化而改变预期的I/O动作。

I/O寄存器和RAM的最主要区别就是I/O操作具有边际效应，而内存操作则没有：内存写操作的唯一结果就是在指定位置存储一个数值；内存读操作则仅仅返回指定位置最后一次写入的数值。由于内存访问速度对CPU的性能至关重要，而且也没有边际效应，所以可用多种方法进行优化，如使用高速缓存保存数值、重新排序读/写指令等。

编译器能够将数值缓存在CPU寄存器中而不写入内存，即使存储数据，读写操作也都能在高速缓存中进行而不用访问物理RAM。无论在编译器一级或是硬件一级，指令的重新排序都有可能发生：一个指令序列如果以不同于程序文本中的次序运行常常能执行得更快，例如在防止RISC处理器流水线的互锁时就是如此。在CISC处理器上，耗时的操作则可以和运行较快的操作并发执行。

在对常规内存进行这些优化的时候，优化过程是透明的，而且效果良好（至少在单处理器系统上是这样），但对I/O操作来说这些优化很可能造成致命的错误，这是因为它们受

到边际效应的干扰，而这却是驱动程序访问I/O寄存器的主要目的。处理器无法预料某些其他进程（在另一个处理器上运行，或在某个I/O控制器中发生的操作）是否会依赖于内存访问的顺序。编译器或CPU可能会自作聪明地重新排序所要求的操作，结果会发生奇怪的错误，并且很难调试。因此，驱动程序必须确保不使用高速缓存，并且在访问寄存器时不发生读或写指令的重新排序。

由硬件自身缓存引起的问题很好解决：只要把底层硬件配置成（可以是自动的或是由Linux初始化代码完成）在访问I/O区域（不管是内存还是端口）时禁止硬件缓存即可。

由编译器优化和硬件重新排序引起的问题的解决办法是，对硬件（或其他处理器）必须以特定顺序执行的操作之间设置内存屏障（memory barrier）。Linux提供了4个宏来解决所有可能的排序问题：

```
#include <linux/kernel.h>
void barrier(void)
```

这个函数通知编译器插入一个内存屏障，但对硬件没有影响。编译后的代码会把当前CPU寄存器中的所有修改过的数值保存到内存中，需要这些数据的时候再重新读出来。对*barrier*的调用可避免在屏障前后的编译器优化，但硬件能完成自己的重新排序。

```
#include <asm/system.h>
void rmb(void);
void read_barrier_depends(void);
void wmb(void);
void mb(void);
```

这些函数在已编译的指令流中插入硬件内存屏障；具体的实现方法是平台相关的。*rmb*（读内存屏障）保证了屏障之前的读操作一定会在后来的读操作执行之前完成。*wmb*保证写操作不会乱序，*mb*指令保证了两者都不会。这些函数都是*barrier*的超集。

*read_barrier_depends*是一种特殊的、弱一些的读屏障形式。我们知道，*rmb*避免屏障前后的所有读取指令被重新排序，而*read_barrier_depends*仅仅阻止某些读取操作的重新排序，这些读取依赖于其他读取操作返回的数据。它和*rmb*的区别很微妙，而且并不是所有的架构上都存在这个函数。除非读者能够正确理解它们之间的差别，并且有理由相信完整的读取屏障会导致额外的性能消耗，否则就应该始终使用*rmb*。

```
void smp_rmb(void);
void smp_read_barrier_depends(void);
void smp_wmb(void);
void smp_mb(void);
```

上述屏障宏版本也插入硬件屏障，但仅仅在内核针对SMP系统编译时有效；在单处理器系统上，它们均会被扩展为上面那些简单的屏障调用。

设备驱动程序中使用内存屏障的典型形式如下：

```
writel(dev->registers.addr, io_destination_address);
writel(dev->registers.size, io_size);
writel(dev->registers.operation, DEV_READ);
wmb();
writel(dev->registers.control, DEV_GO);
```

在这个例子中，最重要的是要确保控制某特定操作的所有设备寄存器一定要在操作开始之前已被正确设置。其中的内存屏障会强制写操作以要求的顺序完成。

因为内存屏障会影响系统性能，所以应该只用于真正需要的地方。不同类型的内存屏障对性能的影响也不尽相同，所以最好尽可能使用最符合需要的特定类型。例如，在x86体系架构上，由于处理器之外的写入操作不会被重新排序，因此，*wmb()*不会做任何事情。但是，读取会重新排序，所以*mb()*就会比*wmb()*慢一些。

值得注意的是，大多数处理同步的内核原语，如自旋锁和atomic_t操作，也能作为内存屏障使用。同时还需注意，某些外设总线（比如PCI总线）存在自身的高速缓存问题，我们将在后面的章节中讨论相关问题。

在某些体系架构上，允许把赋值语句和内存屏障进行合并以提高效率。内核提供了几个执行这种合并的宏，在默认情况下，这些宏的定义如下：

```
#define set_mb(var, value)  do {var = value; mb();}  while 0
#define set_wmb(var, value) do {var = value; wmb();} while 0
#define set_rmb(var, value) do {var = value; rmb();} while 0
```

在适当的地方，*<asm/system.h>*中定义的这些宏可以利用体系架构特有的指令更快地完成任务。注意，只有小部分体系架构定义了*set_rmb*宏（使用do...while来构造宏是标准C的惯用方法，这种方法保证扩展后的宏可在所有上下文环境中当作一个正常C语句来执行）。

使用 I/O 端口

I/O端口是驱动程序与许多设备的之间通信方式 —— 至少在部分时间是这样。本节讲解了使用I/O端口的不同函数，另外也涉及到一些可移植性问题。

I/O 端口分配

读者会想到，在尚未取得对这些端口的独占访问之前，我们不应对这些端口进行操作。内核为我们提供了一个注册用的接口，它允许驱动程序声明自己需要操作的端口。该接口的核心函数是 *request_region*:

```
#include <linux/ioport.h>
struct resource *request_region(unsigned long first, unsigned long n,
                                const char *name);
```

这个函数告诉内核，我们要使用起始于first的n个端口。参数name应该是设备的名称。如果分配成功，则返回非NULL值。如果 *request_region* 返回NULL，那么我们就不能使用这些期望的端口。

所有的端口分配可从 */proc/ioports* 中得到。如果我们无法分配到需要的端口集合，则可以通过这个 */proc* 文件得知哪个驱动程序已经分配了这些端口。

如果不再使用某组I/O端口（可能在卸载模块时），则应该使用下面的函数将这些端口返回给系统:

```
void release_region(unsigned long start, unsigned long n);
```

下面的函数允许驱动程序检查给定的I/O端口集是否可用:

```
int check_region(unsigned long first, unsigned long n);
```

这里，如果给定的端口不可用，则返回值是负的错误代码。我们不赞成使用这个函数，因为它的返回值并不能确保分配是否能够成功，这是因为，检查和其后的分配并不是原子的操作。我们在这里列出这个函数，是因为仍有一些驱动程序在使用它，但是我们应该始终使用 *request_region*，因为这个函数执行了必要的锁定，以确保分配过程以安全、原子的方式完成。

操作 I/O 端口

当驱动程序请求了需要使用的I/O端口范围后，必须读取和/或写入这些端口。为此，大

多数硬件都会把8位、16位和32位的端口区分开来。它们不能像访问系统内存那样混淆使用（注2）。

因此，C语言程序必须调用不同的函数来访问大小不同的端口。如前一节所述，那些只支持内存映射的I/O寄存器的计算机体系架构通过把I/O端口地址重新映射到内存地址来伪装端口I/O，并且为了易于移植，内核对驱动程序隐藏了这些细节。Linux内核头文件中（在与体系架构相关的头文件<*asm/io.h*>中）定义了如下一些访问I/O端口的内联函数。

```
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
```

字节（8位宽度）读写端口。port参数在一些平台上被定义为unsigned long，而在另一些平台上被定义为unsigned short。不同平台上*inb*返回值的类型也不相同。

```
unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
```

这些函数用于访问16位端口（字宽度）；不能用于S390平台，因为这个平台只支持字节宽度的I/O操作。

```
unsigned inl(unsigned port);
void outl(unsigned longword, unsigned port);
```

这些函数用于访问32位端口。longword参数根据不同平台被定义成unsigned long类型或unsigned int类型。和字宽度I/O一样，“长字”I/O在S390平台上也不可使用。

注意：从现在开始，如果我们使用unsigned而不进一步指定类型信息的话，那么就是在谈论一个与体系架构相关的定义，此时不必关心它的准确特性。这些函数基本是可移植的，因为编译器在赋值时会自动进行强制类型转换（cast）——强制转换成unsigned类型可防止编译时出现的警告信息。只要程序员赋值时注意避免溢出，这种强制类型转换就不会丢失信息。在本章剩余部分将会一直保持这种“不完整的类型定义”的方式。

注意，这里没有定义64位的I/O操作。即使在64位的体系架构上，端口地址空间也只使用最大32位的数据通路。

注2：有时I/O端口和内存一样，例如，可以将两个8位的操作合并成一个16位的操作。如PC的显卡就可以，但一般来说不能认为一定具有这种功能。

在用户空间访问I/O端口

上面这些函数主要是提供给设备驱动程序使用的，但它们也可以在用户空间使用，至少在PC类计算机上可以使用。GNU的C库在*<sys/io.h>*中定义了这些函数。如果要在用户空间代码中使用*inb*及其相关函数，则必须满足下面这些条件：

- 编译该程序时必须带*-O*选项来强制内联函数的展开。
- 必须用*ioperm*或*iopl*系统调用来获取对端口进行I/O操作的权限。*ioperm*用来获取对单个端口的操作权限，而*iopl*用来获取对整个I/O空间的操作权限。这两个函数都是x86平台特有的。
- 必须以root身份运行该程序才能调用*ioperm*或*iopl*（注3）。或者，进程的祖先进程之一已经以root身份获取对端口的访问。

如果宿主平台没有*ioperm*和*iopl*系统调用，则用户空间程序仍然可以使用*/dev/port*设备文件访问I/O端口。不过要注意，该设备文件的含义与平台密切相关，并且除PC平台以外，它几乎没有什么用处。

示例程序*misc-progs/inp.c*和*misc-progs/outp.c*是在用户空间通过命令行读写端口的一个小工具。它们会以多个名字安装（如*inb*、*inw*、*inl*）并且按用户所调用的名字相应地操作字节端口、字端口或双字端口。这些程序在x86平台上使用*ioperm*或者*iopl*，而在其他平台上使用*/dev/port*。

作为尝试，可以将这些程序设置为setuid root，这样，不用显式地获取特权就可以使用硬件了。但请不要在生产用的系统上安装这些setuid程序，因为这种设计本身就是安全漏洞。

串操作

以上的I/O操作都是一次传输一个数据，作为补充，有些处理器上实现了一次传输一个数据序列的特殊指令，序列中的数据单位可以是字节、字或双字。这些指令称为串操作指令，它们执行这些任务时比一个C语言编写的循环语句快得多。下面列出的宏实现了串I/O，它们或者使用一条机器指令实现，或者在没有串I/O指令的平台上使用紧凑循环实现。S390平台上没有定义这些宏。这不会影响可移植性，因为该平台的外设总线不同，通常不会和其他平台使用同样的设备驱动程序。

注3：从技术上说，必须有CAP_SYS_RAWIO的权能，但这与在当前系统上以root身份运行是一样的。

串 I/O 函数的原型如下:

```
void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
```

从内存地址 addr 开始连续读/写 count 数目的字节。只对单一端口 port 读取或写入数据。

```
void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);
```

对一个 16 位端口读/写 16 位数据。

```
void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);
Read or write 32-bit values to a single 32-bit port.
```

对一个 32 位端口读/写 32 位数据。

在使用串 I/O 操作函数时，需要铭记的是：它们直接将字节流从端口中读取或写入。因此，当端口和主机系统具有不同的字节序时，将导致不可预期的结果。使用 *inw* 读取端口将在必要时交换字节，以便确保读入的值匹配于主机的字节序。然而，串函数不会完成这种交换。

暂停式 I/O

在处理器试图从总线上快速传输数据时，某些平台（特别是 i386）会遇到问题。当处理器时钟相比外设时钟快时（比如 ISA）就会出现问题，并且在设备板卡特别慢时表现出来。解决办法是在每条 I/O 指令之后，如果还有其他类似指令，则插入一个小的延迟。在 x86 平台上，这种暂停可通过对端口 0x80 的一条 `out b` 指令实现（通常这样做，但很少使用），或者通过使用忙等待实现。相关细节可参考自己平台上 *asm* 子目录下的 *io.h* 文件。

如果有设备丢失数据的情况，或为了防止出现丢失数据的情况，可以使用暂停式的 I/O 函数来取代通常的 I/O 函数。这些暂停式的 I/O 函数很像前面已经列出的那些 I/O 函数，不同之处是它们的名字用 *_p* 结尾，如 *inb_p*、*outb_p* 等等。在 Linux 支持的大多数平台上都定义了这些函数，不过它们常常扩展为和非暂停式 I/O 同样的代码，因为如果某种体系架构不使用过时的外设总线，就不需要额外的暂停。

平台相关性

由于自身的特性，I/O 指令是与处理器密切相关的。因为它们的工作涉及到处理器移入

移出数据的细节，所以隐藏平台间的差异非常困难。因此，大部分与I/O端口有关的源代码都与平台相关。

回顾前面的函数列表，可以看到有一处不兼容的地方，即数据类型。函数的参数根据各平台体系架构上的不同要相应地使用不同的数据类型。例如，port参数在x86平台（处理器只支持64KB的I/O空间）上定义为unsigned short，但在其他平台上定义为unsigned long，在这些平台上，端口是与内存在同一地址空间内的一些特定区域。

其他一些与平台相关的问题来源于处理器基本结构上的差异，因此也无法避免。因为本书假定读者不会在不了解底层硬件的情况下为特定的系统编写驱动程序，所以不会详细讨论这些差异。下面是内核支持的体系架构可以使用的函数的总结：

IA-32 (x86)

x86_64

该体系架构支持本章提到的所有函数。端口号的类型是unsigned short。

IA-64 (Itanium)

支持所有函数；端口类型是unsigned long（映射到内存）。串操作函数是用C语言实现的。

Alpha

支持所有函数，而I/O端口是映射到内存的。根据不同的Alpha平台上所使用的芯片组的不同，端口I/O操作的实现也有所不同。串操作是用C语言实现的，在文件*arch/alpha/lib/io.c*中定义。端口类型是unsigned long。

ARM

端口映射到内存，支持所有函数；串操作用C语言实现。端口类型是unsigned int。

Cris

该体系架构不支持I/O端口的抽象接口，即使在仿真模式下也是如此，上述各种端口操作被定义为不做任何事情。

M68k

M68k-nommu

端口映射到内存。支持串操作，端口类型是unsigned char *。

MIPS

MIPS64

MIPS端口支持所有函数。因为该处理器不提供机器级的串I/O操作，所以串操作是用汇编语言编写的紧凑循环（tight loop）实现的。端口映射到内存，端口类型是unsigned long。

PA-RISC

支持所有的函数。在基于PCI总线的系统上，端口是int型，而在EISA系统上，端口是unsigned short型，但串操作是例外，它使用unsigned long的端口类型。

PowerPC

PowerPC64

支持所有函数；在32位系统上，端口类型为unsigned char *，在64位系统上，端口类型为unsigned long。

S390

类似于M68k，该平台的头文件只支持字节宽度的端口I/O、不支持串操作。端口类型是字符型（char）指针，映射到内存。

Super-H

端口类型是unsigned int（映射到内存），支持所有函数。

SPARC

SPARC64

和前面一样，I/O空间映射到内存。端口操作函数的port参数类型是unsigned long。

感兴趣的读者可以从*io.h*文件获得更多信息，除了本章所介绍的函数，一些与体系架构相关的函数有时也由该文件定义。不过要注意的是，这些文件阅读起来会比较困难。

值得提及的是，x86家族之外的处理器都不为端口提供独立的地址空间，尽管使用其中几种处理器的机器带有ISA和PCI插槽（两种总线都实现了不同的I/O和内存地址空间）。

此外，一些处理器（特别是早期的Alpha处理器）没有一次传输1或2个字节的指令（注4）。因此，它们的外设芯片通过把端口映射到内存地址空间的特殊地址范围来模拟8位和16位的I/O访问。这样，对同一个端口的*inb*和*inw*指令实现为两个32位的读取不同内存地址的操作。幸好，本章前面介绍的宏的内部实现对驱动程序开发人员隐藏了这些细节，不过这个特点还是很有趣的。想进一步深入研究的读者可以参阅*include/asm-alpha/core_lca.h*中的例子。

注4：单字节I/O操作并没有想像中那么重要，因为这种操作很少发生。为了读/写任意地址空间的单个字节，需要实现一条从寄存器组数据总线低位到外部数据总线任意字节位置的数据通路。这种数据通路在每一次数据传输中都需要额外的逻辑门。不使用这类字节宽度的存取指令可以提升系统的总体性能。

I/O操作在各个平台上执行的细节在对应平台的编程手册中有详细的叙述；也可从Web上下载这些手册的PDF文件。

I/O端口示例

我们用来演示设备驱动程序的端口I/O的示例代码运行于通用的数字I/O端口上，这种端口在大多数计算机平台上都能找到。

数字I/O端口最常见的形式是一个字节宽度的I/O区域，它或者映射到内存，或者映射到端口。当把数值写入到输出区域时，输出引脚上的电平信号随着写入的各位而发生相应变化。从输入区域读取到的数据则是输入引脚各位当前的逻辑电平值。

这类I/O端口的具体实现和软件接口是因系统而异的。大多数情况下，I/O引脚是由两个I/O区域控制的：一个区域中可以选择用于输入和输出的引脚，另一个区域中可以读写实际的逻辑电平。不过有时候情况简单些，每个位不是输入就是输出（不过在这种情况下不能再称为“通用I/O”了）；在所有个人计算机上都能找到的并口就是这样的非通用的I/O端口。我们随后介绍的示例代码要用到这些I/O引脚。

并口简介

因为假定大多数读者使用的都是称为“个人计算机”的x86平台，所以解释一下PC并口的设计思想是必要的。并口也是在我们在个人计算机上运行数字I/O示例代码时选用的外设接口。尽管许多读者手头可能有并口的规范，但为了方便，还是在这里概括一下。

并口的最小配置（不涉及ECP和EPP模式）由3个8位端口组成。PC标准中第一个并口的I/O端口是从地址0x378开始，第二个端口是从地址0x278开始。第一个端口是一个双向的数据寄存器；它直接连接到物理连接器的2～9号引脚上。第二个端口是一个只读的状态寄存器；当并口连接到打印机时，该寄存器报告打印机的状态，如是否在线、缺纸、正忙等等。第三个端口是一个只用于输出的控制寄存器，它的作用之一是控制是否启用中断。

在并行通信中使用的电平信号是标准的TTL电平：0伏和5伏，逻辑阈值大约为1.2伏。我们可以认为端口至少满足标准的TTL LS电流规格，而现代的大部分并口的电流和电压都超过了上述规格的定义。

警告： 并口连接器没有和计算机的内部电路隔离，这一点在试图把逻辑门直接连到端口时很有用。但要注意正确连线；否则在测试自己定制的电路时，很容易烧毁并口。如果担心毁坏主板的话，可以选用可插拔的并行接口。

图9-1说明了并口的位规范。可以读写12个输出位和5个输入位，其中一些位在它们的信号通路上会有逻辑上的翻转。唯一一个不与任何信号引脚有联系的位是2号端口的第4位(0x10)，它启用来自并口的中断。我们将在第十章中的一个中断处理程序中使用它。

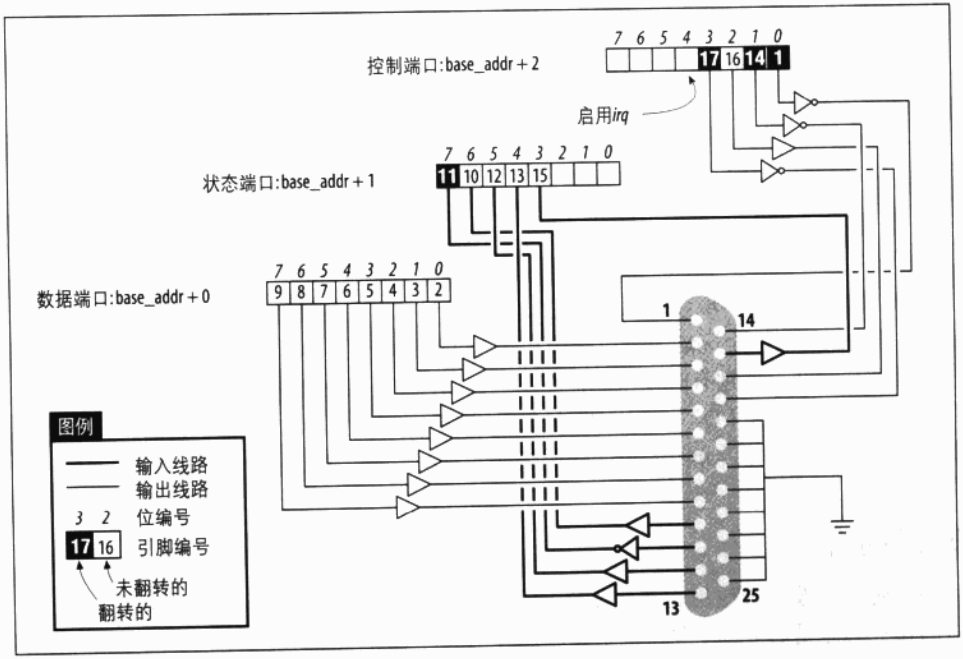


图9-1：并口的引脚

示例驱动程序

下面要介绍的驱动程序叫做short（Simple Hardware Operations and Raw Tests，简单的硬件操作和裸测试）。它所做的就是读写几个8位端口，起始的端口是加载时选定的。默认情况下它使用的就是分配给PC并口的端口范围。每个设备节点（拥有唯一的次设备号）访问一个不同的端口。*short*设备没有任何实际用途，使用它只是为了能用一条指令来从外部对端口进行操作。如果读者不太了解端口I/O，那么可以通过使用*short*来熟悉它，可以测量它传输数据时消耗的时间或者进行其他的测试。

为使*short*在系统上工作，它必须能自由地访问底层硬件设备（默认情况下就是并口），因此不能有其他驱动程序在使用同一设备。现在的大多数Linux发布版本将并口驱动程序作为模块安装，并且只在需要用到的时候才加载，所以一般不会发生争夺I/O地址的问题。不过，如果*short*给出一个“can't get I/O address（无法获得I/O地址）”的错误

(可能在控制台或者系统日志文件中),则说明可能已经有其他驱动程序占用了这个端口。通过检查*/proc/ioports*一般可以找出这是哪个驱动程序。这种情况同样适用于并口之外的其他I/O设备。

从现在开始,为简化讨论,我们所指的设备都是并口。不过也可以在模块加载时通过设置参数base把*short*重定向到其他I/O设备。这样,示例代码可以在任何拥有对数字I/O接口访问权限的Linux平台上运行,这些接口必须是能用*outb*和*inb*进行访问的(尽管实际硬件在除x86之外的所有平台上都是映射到内存的)。在随后的"使用I/O内存"一节中,我们还将展示*short*是如何用于通用的映射到内存的数字I/O的。

为了观察并口连接器上发生了什么,并且如果读者喜欢操作硬件,那么可以焊几个LED到输出引脚上。每个LED都要串联一个1kΩ的电阻到一个接地的引脚上(除非使用的LED已经有内建电阻)。如果将输出引脚接到输入引脚上,就可以产生自己的输入供输入端口读取。

注意,不能仅仅通过把打印机连接到并口来观察送给*short*的数据。因为这个驱动程序只实现了简单的I/O端口访问,不能提供打印机操作数据时所需的握手信号。下一章介绍的示例驱动程序(称为*shortprint*)可驱动并口打印机,但是该驱动程序使用了中断,因此目前还不能介绍这个驱动程序。

如果读者打算将LED焊到D型连接器上来观察并行数据,建议不要使用9号和10号引脚,因为在运行第十章的示例代码时我们要使用这些引脚。

至于*short*,它通过*/dev/short0*读写位于I/O地址base(除非加载时修改,否则就是0x378)的8位端口。*/dev/short1*写入位于`base + 1`的8位端口,依次类推,直到`base + 7`。

*/dev/short0*实际执行的输出操作是一个使用*outb*的紧凑循环。这里还使用了内存屏障指令来确保输出操作会实际执行而不是被优化掉。

```
while (count--) {
    outb(*(ptr++), port);
    wmb();
}
```

可以运行下面的命令来使LED发光:

```
echo  -n "any string"  > /dev/short0
```

每个LED监控输出端口的一个位。注意,只有最后写入的字符数据才会在输出引脚上稳定地保持下来而被观察到。因此,建议将*-n*选项传递给*echo*程序来制止输出字符后的自动换行。

读取端口也使用类似的函数，只是用*inb*代替了*outb*。为了从并口读取“有意义的”值，需要将某个硬件连接到并口连接器的输入引脚上来产生信号。如果没有输入信号，则只会读取到始终是相同字节的无穷输出流。如果选择从输出端口读入，将会取回写入到该端口的最后一个值（对并口和其他大多数普通数字I/O电路都是如此）。因此，不想摆弄烙铁的读者可以运行下面的命令在端口0x378读取当前的输出值：

```
dd if=/dev/short0 bs=1 count=1 | od -t x1
```

为了示范所有I/O指令的使用，每个*short*设备都提供了3个变种：*/dev/short0*执行的是上面的循环；*/dev/short0p*使用了*outb_p*和*inb_p*来替代前者使用的“较快的”函数，*/dev/short0s*使用了串指令。这样的设备共有8个，从*short0*到*short7*。PC并口只有三个端口，如果读者使用了其他不同的I/O设备进行测试，就可能需要更多的端口。

虽然*short*驱动程序只完成了最低限度的硬件控制，但这对演示I/O端口指令的使用已经足够了。感兴趣的读者可以参阅*parport*和*parport_pc*两个模块的源代码，看看为支持使用并口的设备（打印机、磁带备份、网络接口）所需的复杂工作。

使用I/O内存

除了x86上普遍使用的I/O端口之外，和设备通信的另一种主要机制是通过使用映射到内存的寄存器或设备内存。这两种都称为I/O内存，因为寄存器和内存的差别对软件是透明的。

I/O内存仅仅是类似RAM的一个区域，在那里处理器可以通过总线访问设备。这种内存有很多用途，比如存放视频数据或以太网数据包，也可以用来实现类似I/O端口的设备寄存器（也就是说，对它们的读写也存在边际效应）。

访问I/O内存的方法和计算机体系架构、总线以及正在使用的设备有关，不过原理都是相同的。本章主要讨论ISA和PCI内存，同时也试着介绍一些通用的知识。尽管这里介绍了对PCI内存的访问，但关于PCI的详细讨论将放到第十二章中进行。

根据计算机平台和所使用总线的不同，I/O内存可能是、也可能不是通过页表访问的。如果访问是经由页表进行的，内核必须首先安排物理地址使其对设备驱动程序可见（这通常意味着在进行任何I/O之前必须先调用*ioremap*）。如果访问无需页表，那么I/O内存区域就非常类似于I/O端口，可以使用适当形式的函数读写它们。

不管访问I/O内存时是否需要调用*ioremap*，都不鼓励直接使用指向I/O内存的指针。尽管（如在“I/O端口和I/O内存”一节中介绍的那样）I/O内存在硬件一级像普通RAM一样寻址，但在“I/O寄存器和常规内存”一节中描述过的需要额外小心的内容中，我

们不建议使用普通的指针。相反，使用包装函数访问I/O内存，这一方面在所有平台上都是安全的，另一方面，在可以直接对指针指向的内存区域执行操作的时候，这些函数是经过优化的。

因此，即使在x86上直接使用指针（现在）可以工作（而不是使用适当的宏），这种做法也会影响驱动程序的可移植性和可读性。

I/O内存分配和映射

在使用之前，必须首先分配I/O内存区域。用于分配内存区域的接口（在*<linux/ioport.h>*中定义）如下所示：

```
struct resource *request_mem_region(unsigned long start, unsigned long len,
                                    char *name);
```

该函数从`start`开始分配`len`字节长的内存区域。如果成功，返回非NULL指针；否则返回NULL值。所有的I/O内存分配情况均可从*/proc/iomem*得到。

不再使用已分配的内存区域时，使用下面的接口释放：

```
void release_mem_region(unsigned long start, unsigned long len);
```

下面是用来检查给定的I/O内存区域是否可用的老函数：

```
int check_mem_region(unsigned long start, unsigned long len);
```

但是，和*check_region*一样，这个函数不安全，应避免使用。

分配I/O内存并不是访问这些内存之前需要完成的唯一步骤，我们还必须确保该I/O内存对内核而言是可访问的。获取I/O内存并不仅仅意味着可引用对应的指针；在许多系统上，I/O内存根本不能通过这种方式直接访问。因此，我们必须首先建立映射。映射的建立由*ioremap*函数完成，我们在第八章的“vmalloc及其辅助函数”一节中介绍过这个函数。该函数专用于为I/O内存区域分配虚拟地址。

一旦调用*ioremap*（以及*iounmap*）之后，设备驱动程序即可访问任意的I/O内存地址了，而无论I/O内存地址是否直接映射到虚拟地址空间。但要记住，由*ioremap*返回的地址不应直接引用，而应该使用内核提供的accessor函数。在我们介绍这些函数之前，首先复习一下*ioremap*的原型并介绍一些先前章节中跳过的细节内容。

我们根据以下的定义来调用*ioremap*函数：

```
#include <asm/io.h>
void *ioremap(unsigned long phys_addr, unsigned long size);
```

```
void *ioremap_nocache(unsigned long phys_addr, unsigned long size);
void iounmap(void * addr);
```

首先读者会注意到新的函数：*ioremap_nocache*。我们并没有在第八章介绍这个函数，因为该函数的功能和硬件相关。内核头文件中有如下一段解释："如果某些控制寄存器也在此类区域，而不希望出现写入组合或者读取高速缓存的话，则可使用该函数。"实际上，在大多数计算机平台上，该函数的实现和*ioremap*相同：当所有I/O内存已属于非缓存地址时，就没有必要实现*ioremap*的独立的、非缓存版本。

访问I/O内存

在某些平台上，我们可以将*ioremap*的返回值直接当作指针使用。但是，这种使用不具有可移植性，而内核开发者正在致力于减少这类使用。访问I/O内存的正确方法是通过一组专用于此目的的函数（在*<asm/io.h>*中定义）。

要从I/O内存中读取，可使用下面函数之一：

```
unsigned int ioread8(void *addr);
unsigned int ioread16(void *addr);
unsigned int ioread32(void *addr);
```

其中，addr应该是从*ioremap*获得的地址（可能包含一个整数偏移量）；返回值则是从给定I/O内存读取到的值。

还有一组用于写入I/O内存的类似函数集如下：

```
void iowrite8(u8 value, void *addr);
void iowrite16(u16 value, void *addr);
void iowrite32(u32 value, void *addr);
```

如果必须在给定的I/O内存地址处读/写一系列的值，则可使用上述函数的重复版本：

```
void ioread8_rep(void *addr, void *buf, unsigned long count);
void ioread16_rep(void *addr, void *buf, unsigned long count);
void ioread32_rep(void *addr, void *buf, unsigned long count);
void iowrite8_rep(void *addr, const void *buf, unsigned long count);
void iowrite16_rep(void *addr, const void *buf, unsigned long count);
void iowrite32_rep(void *addr, const void *buf, unsigned long count);
```

上述函数从给定的buf向给定的addr读取或写入count个值。注意，count以被写入的数据大小为单位表示，比如，*ioread32_rep*从addr中读取count个32位的值到buf中。

上面给出的函数均在给定的addr处执行所有的I/O操作。如果我们要在一块I/O内存上执行操作，则可以使用下面的函数之一：

```
void memset_io(void *addr, u8 value, unsigned int count);
void memcpy_fromio(void *dest, void *source, unsigned int count);
void memcpy_toio(void *dest, void *source, unsigned int count);
```

上述函数和C函数库的对应函数功能一致。

如果读者阅读内核源代码，可能会遇到一组老的I/O内存函数。这些函数仍能工作，但不鼓励在新的代码中使用这些函数。主要原因是因为这些函数不执行类型检查，因此其安全性较差。这些函数（宏）的原型如下：

```
unsigned readb(address);
unsigned readw(address);
unsigned readl(address);
```

这些宏用来从I/O内存检索8位、16位和32位的数据。

```
void writeb(unsigned value, address);
void writew(unsigned value, address);
void writel(unsigned value, address);
```

类似前面的函数，这些函数（宏）用来写8位、16位和32位的数据项。

一些64位平台还提供了*readq*和*writeq*，用于PCI总线上的4字（8字节）内存操作。这个4字（quad-word）的命名是个历史遗留问题，那时候所有的处理器都只有16位的字。实际上，现在把32位的数值叫做*L*（长字）已经是不正确的了，不过如果对一切都重新命名，只会把事情搞得更复杂。

像I/O内存一样使用端口

某些硬件具有一种有趣的特性：某些版本使用I/O端口，而其他版本使用I/O内存。导出给处理器的寄存器在两种情况下都是一样的，但访问方法却不同。为了让处理这类硬件的驱动程序更加易于编写，也为了最小化I/O端口和内存访问之间的表面区别，2.6内核引入了*ioport_map*函数：

```
void *ioport_map(unsigned long port, unsigned int count);
```

该函数重新映射count个I/O端口，使其看起来像I/O内存。此后，驱动程序可在该函数返回的地址上使用*ioread8*及其同类函数，这样就不必理会I/O端口和I/O内存之间的区别了。

当不再需要这种映射时，需要调用下面的函数来撤消：

```
void ioport_unmap(void *addr);
```

这些函数使得I/O端口看起来像内存。但需要注意的是，在重新映射之前，我们必须通过*request_region*来分配这些I/O端口。

为I/O内存重用short

前面介绍的*short*示例模块访问的是I/O端口，它也可以访问I/O内存。为此必须在加载时通知它使用I/O内存，另外还要修改base的地址以使其指向I/O区域。

例如，我们用下列命令在一块MIPS开发板上点亮调试用的LED:

```
mips.root# ./short_load use_mem=1 base=0xb7ffffc0
mips.root# echo -n 7 > /dev/short0
```

在*short*中使用I/O内存和使用I/O端口是一样的。

下面的代码段说明了*short*写入内存区域时使用的循环:

```
while (count--) {
    iowrite8(*ptr++, address);
    wmb();
}
```

注意，这里使用了写内存屏障。因为在许多体系架构上*iowrite8*会转化成一个直接赋值语句，所以为确保写操作按照预期的顺序执行，使用内存屏障是必要的。

*short*使用*inb*和*outb*来完成相应的工作。但是，修改*short*并使用*ioport_map*以便将I/O端口映射为I/O内存的工作非常直接（这将大大简化其余的代码），因此留给读者来完成。

1MB地址空间之下的ISA内存

最广为人知的I/O内存区之一就是个人计算机上的ISA内存段。它的内存范围在640KB（0xA0000）到1MB（0x100000）之间，因此它正好出现在常规系统RAM的中间。这种地址的安排看上去可能有点奇怪，但因为这个设计决策是20世纪80年代早期作出的，在当时看来没有人会用到640KB以上的内存。

这个内存段属于非直接映射一类的内存（注5）。如此可以利用*short*模块在该内存段中读写几个字节，前面已介绍过，在加载模块时要设置use_mem标志。

注5：实际上并非完全如此。因为该内存范围很小而且使用频繁，所以内核在启动时就建立了访问这些地址的页表。但是，访问它们用的虚拟地址和实际物理地址并不相同，所以无论如何都要使用*ioremap*。

尽管ISA I/O内存只存在于x86类的计算机上，但我们还是介绍一下，并附以一个示例程序。

本章不讨论PCI内存，因为它是I/O内存中最“干净”的一种：只要知道了物理地址，就能简单地重映射并访问这些内存。PCI I/O内存的“问题”在于，它不适合于用作本章的示例，因为无法预先知道PCI内存会映射到哪一段物理地址，也就不知道访问这些地址段是否安全。这里选择讲解ISA内存段，是因为它虽然不那么“干净”，但更适合于运行示例代码。

为了示范对ISA内存的访问，我们要用到另一个有点“愚笨”的小模块（是示例源代码的一部分）。实际上这个模块就叫做*silly*，是“Simple Tool for Unloading and Printing ISA Data（卸载及打印ISA数据的简单工具）”的缩写。

这个模块补充了*short*的功能，它可以访问整个384 KB的内存空间，还演示了所有不同的I/O函数。该模块包括四个使用不同的数据传输函数来完成相同任务的设备节点。*silly*设备就像I/O内存之上的一个窗口，与*/dev/mem*的工作有些类似。对该设备可以读、写数据或*lseek*到一个任意的I/O内存地址。

因为*silly*提供对ISA内存的访问，所以启动它时必须把物理ISA地址映射到内核虚拟地址中。在较早的Linux内核中，只需简单地把要用的ISA地址赋值给一个指针，然后直接解析它就可以了。但在现在的内核中，必须配合虚拟内存系统工作，首先重新映射该地址段。这种映射是由*ioremap*完成的，这在前面讲解*short*时已经介绍过了：

```
#define ISA_BASE    0xA0000
#define ISA_MAX     0x100000  /* 用于一般的内存访问 */

    /* 下面这条语句出现在 silly_init 中 */
    io_base = ioremap(ISA_BASE, ISA_MAX - ISA_BASE);
```

*ioremap*返回一个指针值，以供*ioread8*或其他在“访问I/O内存”一节中介绍的函数使用。

现在回顾示例代码中这些函数是如何使用的。*/dev/sillyb*的次设备号是0，通过*ioread8*和*iowrite8*访问I/O内存。下面的代码展示了读操作的实现，其中地址段0xA0000～0xFFFFF作为0～0x5FFFF段的一个虚拟文件对待。*read*函数中包括一个switch语句来处理不同的访问模式。这里是*sillyb*的*case*语句：

```
case M_8:
  while (count) {
      *ptr = ioread8(add);
      add++;
      count--;
      ptr++;
```

```
    }
    break;
```

下面的两个设备是*/dev/sillyw*（次设备号为1）和*/dev/sillyl*（次设备号为2）。它们和*/dev/sillyb*差不多，只不过分别使用了16位和32位的函数。下面是*sillyl*的*write*实现，也是switch语句中的一部分：

```
case M_32:
  while (count >= 4) {
      iowrite8(*(u32 *)ptr, add);
      add += 4;
      count -= 4;
      ptr += 4;
  }
  break;
```

最后一个设备是*/dev/sillycp*（次设备号为3），它使用*memcpy_*io*函数完成相同的任务。它的*read*实现的核心部分如下：

```
case M_memcpy:
  memcpy_fromio(ptr, add, count);
  break;
```

因为使用了*ioremap*来提供对ISA内存区的访问，故卸载*silly*模块时必须调用*iounmap*：

```
iounmap(io_base);
```

isa_readb及相关函数

看看内核源代码，可以发现一组例程，它们的名字类似于*isa_readb*。实际上，上面描述的每个函数都有一个等价的以*isa_*开头的函数。这些函数提供了一种不需要单独的*ioremap*步骤就能访问ISA内存的方法。不过内核开发人员解释说，这些函数只是暂时性的，用于帮助移植驱动程序，将来它们会消失，所以最好避免使用这些函数。

快速参考

本章引入下列与硬件管理有关的符号：

```
#include <linux/kernel.h>
void barrier(void)
```

这个“软件”内存屏障要求编译器考虑执行到该指令时所有的内存易变性。

```
#include <asm/system.h>
void rmb(void);
void read_barrier_depends(void);
void wmb(void);
void mb(void);
```

硬件内存屏障。要求CPU（和编译器）执行该指令时检查所有必需的内存读、写（或二者兼有）已经执行完毕。

```
#include <asm/io.h>
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
unsigned inl(unsigned port);
void outl(unsigned doubleword, unsigned port);
```

这些函数用于读和写I/O端口。如果用户空间的程序有访问端口的权限，则也可以调用这些函数。

```
unsigned inb_p(unsigned port);
...
```

如果I/O操作之后需要一小段延时，可以用上面介绍的函数的6个暂停式的变体。这些暂停式的函数都以 *_p* 结尾。

```
void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);
void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);
```

这些“串操作函数”为输入端口与内存区之间的数据传输做了优化。这类传输是通过对同一端口连续读/写count次实现的。

```
#include <linux/ioport.h>
struct resource *request_region(unsigned long start, unsigned long len,
  char *name);
void release_region(unsigned long start, unsigned long len);
int check_region(unsigned long start, unsigned long len);
```

为I/O端口分配资源的函数。*check_* 函数在成功时返回0，出错时返回负值，但我们不建议使用该函数。

```
struct resource *request_mem_region(unsigned long start, unsigned long len,
  char *name);
void release_mem_region(unsigned long start, unsigned long len);
int check_mem_region(unsigned long start, unsigned long len);
```

这些函数处理对内存区域的资源分配。

```
#include <asm/io.h>
void *ioremap(unsigned long phys_addr, unsigned long size);
void *ioremap_nocache(unsigned long phys_addr, unsigned long size);
void iounmap(void *virt_addr);
```

ioremap 把一个物理地址范围重新映射到处理器的虚拟地址空间，以供内核使用。*iounmap* 用来解除这个映射。

```
#include <asm/io.h>
unsigned int ioread8(void *addr);
unsigned int ioread16(void *addr);
unsigned int ioread32(void *addr);
void iowrite8(u8 value, void *addr);
void iowrite16(u16 value, void *addr);
void iowrite32(u32 value, void *addr);
```

用来访问 I/O 内存的函数。

```
void ioread8_rep(void *addr, void *buf, unsigned long count);
void ioread16_rep(void *addr, void *buf, unsigned long count);
void ioread32_rep(void *addr, void *buf, unsigned long count);
void iowrite8_rep(void *addr, const void *buf, unsigned long count);
void iowrite16_rep(void *addr, const void *buf, unsigned long count);
void iowrite32_rep(void *addr, const void *buf, unsigned long count);
```

I/O 内存访问原语的“重复”版本。

```
unsigned readb(address);
unsigned readw(address);
unsigned readl(address);
void writeb(unsigned value, address);
void writew(unsigned value, address);
void writel(unsigned value, address);
memset_io(address, value, count);
memcpy_fromio(dest, source, nbytes);
memcpy_toio(dest, source, nbytes);
```

也是用来访问 I/O 内存的函数，但老一些且不安全。

```
void *ioport_map(unsigned long port, unsigned int count);
void ioport_unmap(void *addr);
```

如果驱动程序作者希望将I/O端口作为I/O内存一样进行操作，则可将这些端口传递给*ioport_map*函数。不再使用这种映射时，应该使用*ioport_unmap*函数解除映射。

第十章

中断处理



尽管有些设备仅通过它们的I/O寄存器就可以得到控制，但现实中的大部分设备却要比这复杂一些。设备需要与外部世界打交道，比如旋转的磁盘、绕卷的磁带、远距离连接的电缆等等。这些设备的许多工作通常是在与处理器完全不同的时间周期内完成的，并且总是要比处理器慢。这种让处理器等待外部事件的情况总是不能令人满意，所以必须有一种方法可以让设备在产生某个事件时通知处理器。

这种方法就是中断。一个“中断”仅仅是一个信号，当硬件需要获得处理器对它的关注时，就可以发送这个信号。Linux处理中断的方式很大程度上与它在用户空间处理信号是一样的。在大多数情况下，一个驱动程序只需要为它自己设备的中断注册一个处理例程，并且在中断到达时进行正确的处理。当然，这个过程看似简单，但还是有一些复杂性的。需要特别指出的是，随着中断处理例程运行方式的不同，它们所能执行的动作将会受到不同的限制。

如果没有一个真正的硬件设备产生中断，就很难示范中断的使用方法。因而，本章中的示例代码利用并口来产生中断。我们将使用上一章的*short*模块来示范，作一些小的改动就可以通过并口来产生中断并处理中断。模块的名字*short*实际是指*short int*（有点像C语言），提醒我们这个模块要处理中断。

但是在我们深入讨论这个主题之前，有一点值得关注。从本质上讲，中断处理例程和其他代码并发运行，这样，这些处理例程会不可避免地引起并发问题，并竞争数据结构和硬件。如果读者已经阅读了第五章的相关内容，则应该能够理解这里的意思，但还是建议读者再次阅读第五章，以便有更深入的理解。对并发控制技术的透彻理解对处理中断来讲非常重要。

准备并口

尽管并行接口很简单，但它也可以触发中断。打印机就是利用这种能力来通知*lp*驱动程序它已经准备好接受缓冲区中的下一个字符的。

就像大多数设备一样，在没有设定产生中断之前，并口是不会产生中断的；并口的标准规定设置端口2（0x37a、0x27a或者其他端口）的第4位将启用中断报告。*short*模块在初始化的时候调用*outb*来设置这个位。

在中断处于启用状态时，每当引脚10（所谓的ACK位）的电平发生从低到高改变时，并口就会产生一个中断，在没有把打印机连到端口上的情况下，强制接口产生中断的最简单的方法是连接并口连接器的9脚和10脚。将一根短电线插入系统后面并口连接器上的对应的孔，就可以连接这两个引脚。并口的引脚已在图9-1中说明。

引脚9是并口数据字节中的最高位。如果将二进制数据写入*/dev/short0*，就会引发几个中断，将ASCII码文本写入端口则不会产生中断，因为此时没有设置这个最高位。

如果读者手头有一台打印机，并且想要避免焊接电线，则可以运行本章后面针对真实打印机的中断处理例程。注意，我们将要介绍的探测函数依赖于在引脚9和10之间适当的跳线，因此，在使用这些代码做探测试验的时候需要它。

安装中断处理例程

如果读者确实想“看到”产生的中断，那么仅仅通过向硬件设备写入是不够的，还必须要在系统中安装一个软件处理例程。如果没有通知Linux内核等待用户的中断，那么内核只会简单应答并忽略该中断。

中断信号线是非常珍贵且有限的资源，尤其是在系统上只有15根或16根中断信号线时更是如此。内核维护了一个中断信号线的注册表，该注册表类似于I/O端口的注册表。模块在使用中断前要先请求一个中断通道（或者中断请求IRQ），然后在使用后释放该通道。我们将会在后面看到，在很多场合下，模块也希望可以和其他的驱动程序共享中断信号线。下列在头文件*<linux/sched.h>*中声明的函数实现了该接口：

```
int request_irq(unsigned int irq,
                irqreturn_t (*handler)(int, void *, struct pt_regs *),
                unsigned long flags,
                const char *dev_name,
                void *dev_id);

void free_irq(unsigned int irq, void *dev_id);
```

通常，从*request_irq*函数返回给请求函数的值为0时表示申请成功，为负值时表示错误码。函数返回-EBUSY表示已经有另一个驱动程序占用了你要请求的中断信号线。这些函数的参数如下：

`unsigned int irq`
这是要申请的中断号。

`irqreturn_t (*handler)(int, void *, struct pt_regs *)`
这是要安装的中断处理函数指针。我们会在本章的后面部分讨论这个函数的参数含义。

`unsigned long flags`
如读者所想，这是一个与中断管理有关的位掩码选项（将在后面描述）。

`const char *dev_name`
传递给*request_irq*的字符串，用来在*/proc/interrupts*中显示中断的拥有者（参见下节）。

`void *dev_id`
这个指针用于共享的中断信号线。它是唯一的标识符，在中断信号线空闲时可以使用它，驱动程序也可以使用它指向驱动程序自己的私有数据区（用来识别哪个设备产生中断）。在没有强制使用共享方式时，dev_id可以被设置为NULL，总之用它来指向设备的数据结构是一个比较好的思路。我们会在本章后面的“实现处理例程”一节中看到dev_id的实际应用。

可以在flags中设置的位如下所示：

SA_INTERRUPT
当该位被设置时，表明这是一个“快速”的中断处理例程。快速处理例程运行在中断的禁用状态下（更详细的主题将在本章后面的“快速和慢速处理例程”一节中讨论）。

SA_SHIRQ
该位表示中断可以在设备之间共享。共享的概念将在本章后面的“中断共享”一节描述。

SA_SAMPLE_RANDOM
该位指出产生的中断能对*/dev/random*设备和*/dev/urandom*设备使用的熵池（entropy pool）有贡献。从这些设备读取，将会返回真正的随机数，从而有助于应用软件选择用于加密的安全密钥。这些随机数是从一个熵池中得到的，各种随机事件都会对该熵池作出贡献，如果读者的设备以真正随机的周期产生中断，就应该设

置该标志位。另一方面，如果中断是可预期的（例如，帧捕捉卡的垂直消隐），就不值得设置这个标志位 —— 它对系统的熵没有任何贡献。能受到攻击者影响的设备不应该设置该位，例如，网络驱动程序会被外部的事件影响到预定的数据包的时间周期，因而也不会对熵池有贡献，更详细的信息请参见*drivers/char/random.c*文件中的注释。

中断处理例程可在驱动程序初始化时或者设备第一次打开时安装。虽然在模块的初始化函数中安装中断处理例程看起来是个好主意，但实际上并非如此。因为中断信号线的数量是非常有限的，我们不想肆意浪费。计算机拥有的设备通常要比中断信号线多得多，如果一个模块在初始化时请求了IRQ，那么即使驱动程序只是占用它而从未使用，也将会阻止任意一个其他的驱动程序使用该中断。而在设备打开的时候申请中断，则可以共享这些有限的资源。

这种情况很可能出现，例如，在运行一个与调制解调器共用同一中断的帧捕捉卡驱动程序时，只要不同时使用这两个设备就可以共享同一中断。用户在系统启动时装载特殊的设备模块是一种普遍做法，即使该设备很少使用。数据捕捉卡可能会和第二个串口使用相同的中断，我们可以在捕获数据时，避免使用调制解调器连接到互联网服务供应商（ISP），但是如果为了使用调制解调器而不得不卸载一个模块，总是令人不快的。

调用*request_irq*的正确位置应该是在设备第一次打开、硬件被告知产生中断之前。调用*free_irq*的位置是最后一次关闭设备、硬件被告知不用再中断处理器之后。这种技术的缺点是必须为每个设备维护一个打开计数，这样我们才能知道什么时候可以禁用中断。

尽管我们已经讨论了不应该在装载模块时调用*request_irq*，但*short*模块还是在装载时请求了它的中断信号线，这样做的方便之处是，我们可以直接运行测试程序，而不需要额外运行其他的进程来保持设备的打开状态。因此，*short*在它自己的初始化函数（*short_init*）中请求中断，而不是像真正的设备驱动那样在*short_open*中请求中断。

下面这段代码要请求的中断是`short_irq`，对这个变量的实际赋值操作（例如，决定使用哪个IRQ）会在后面给出，因为它与当前的讨论无关。`short_base`是并口使用的I/O地址空间的基地址；向并口的2号寄存器写入，可以启用中断报告。

```
if (short_irq >= 0) {
    result = request_irq(short_irq, short_interrupt,
            SA_INTERRUPT, "short", NULL);
   if (result) {
        printk(KERN_INFO "short: can't get assigned irq %i\n",
                short_irq);
        short_irq = -1;
    }
```

```
        else { /* 真正启用中断 —— 假定这是一个并口 */
            outb(0x10,short_base+2);
        }
    }
```

读者可从代码看出，已经安装的中断处理例程是一个快速的处理例程(SA_INTERRUPT)，不支持中断共享（没有设置 SA_SHIRQ），并且对系统的熵（也没有设置 SA_SAMPLE_RANDOM）没有贡献。最后，代码执行 *outb* 调用来启用并口的中断报告。

值得指出的是，i386 和 x86_64 体系架构定义了如下函数，用于查询某个中断线是否可用：

```
int can_request_irq(unsigned int irq, unsigned long flags);
```

如果能够成功分配给定的中断，则该函数返回非零值。但要注意，在调用 *can_request_irq* 和 *request_irq* 之间，始终可能发生一些事情来改变现状。

/proc 接口

当硬件的中断到达处理器时，一个内部计数递增，这为检查设备是否按预期工作提供了一种方法，产生的中断报告显示在文件 */proc/interrupts* 中。下面是一个双处理器的 Pentium 系统启动几天后该文件的快照：

```
root@montalcino:/bike/corbet/write/ldd3/src/short# m /proc/interrupts
           CPU0       CPU1
  0:    4848108         34    IO-APIC-edge  timer
  2:          0          0          XT-PIC  cascade
  8:          3          1    IO-APIC-edge  rtc
 10:       4335          1   IO-APIC-level  aic7xxx
 11:       8903          0   IO-APIC-level  uhci_hcd
 12:         49          1    IO-APIC-edge  i8042
NMI:          0          0
LOC:    4848187    4848186
ERR:          0
MIS:          0
```

第一列是IRQ号，其中明显缺少一些中断，这说明该文件只会显示那些已经安装了中断处理例程的中断。例如，第一个串口（使用中断号4）没有显示，说明我们没有使用调制解调器。实际上，即使早些时候已经使用过调制解调器，而如果在文件快照的时候没有使用的话，也不会出现在这个文件中。串口驱动程序具有良好的行为，在设备被关闭的时候会释放它们的中断处理例程。

文件 */proc/interrupts* 给出了已经发送到系统上每一个 CPU 的中断数量。正如读者能从

输出中看到的，Linux内核通常会在第一个CPU上处理中断，以便最大化缓存的本地性（注1）。最后两列给出了处理中断的可编程中断控制器（驱动程序作者不需要关心该控制器）信息，以及注册了中断处理例程的设备名称（这和传递给*request_irq*的dev_name参数一样）。

*/proc*树结构中还包含另一个与中断相关的文件，即*/proc/stat*。你有时会发现某个文件很有用，有时又会更喜欢使用另外的文件。*/proc/stat*记录了一些系统活动的底层统计信息，包括（但不仅限于）从系统启动开始接收到的中断数量，*stat*文件的每行都以一个字符串开始，它是这行的关键字。intr标记正是我们需要的，下列（被截断和分行）快照是在前一个快照不久之后获得的：

```
intr 5167833 5154006 2 0 2 4907 0 2 68 4 0 4406 9291 50 0 0
```

第一个数是所有中断的总数，而其他的每个数都代表一个单独的IRQ信号线，从中断0开始。这个快照显示4号中断已经发生了4907次，尽管当前并没有安装它的处理例程。如果正在测试的驱动程序在每次打开、关闭设备的周期内请求和释放中断的话，读者就会发现*/proc/stat*比*/proc/interrupts*更有用。

两个文件的另一个不同之处是*interrupts*文件不依赖于体系结构，而*stat*文件则是依赖的：字段的数量依赖于内核之下的硬件。可用的中断数量从SPARC体系结构上的15个，到IA-64结构和一些其他系统上的256个之间变化。值得注意的是，当前x86体系结构上定义的中断数量是224个，不是读者猜测的16个；这可以从头文件*include/asm-386/irq.h*中得到解释，它取决于Linux使用的体系结构的限制而不是特定实现的限制（像16个中断源的老式PC中断控制器）。

下面是文件*/proc/interrupts*在一个IA-64系统上的快照。正如读者看见的，除了将常见中断源通过不同的硬件路由之外，该输出和上面32位系统给出的结果非常相似。

```
           CPU0       CPU1
  27:       1705      34141  IO-SAPIC-level  qla1280
  40:          0          0           SAPIC  perfmon
  43:        913       6960  IO-SAPIC-level  eth0
  47:      26722        146  IO-SAPIC-level  usb-uhci
  64:          3          6   IO-SAPIC-edge  ide0
  80:          4          2   IO-SAPIC-edge  keyboard
  89:          0          0   IO-SAPIC-edge  PS/2 Mouse
 239:    5606341    5606052           SAPIC  timer
 254:      67575      52815           SAPIC  IPI
 NMI:         0          0
 ERR:         0
```

注1：　虽然某些大型系统会显式地使用中断平衡方案在系统范围内传播中断。

自动检测IRQ号

驱动程序初始化时，最迫切的问题之一就是如何决定设备将要使用哪条IRQ信号线。驱动程序需要这个信息以便正确地安装处理例程，尽管程序员可以要求用户在装载时指定中断号，但这不是一个好习惯，因为大部分时间用户不知道这个中断号，或者是因为用户没有配置跳线或者是因为设备是无跳线的。因此，中断号的自动检测对于驱动程序可用性来说是一个基本要求。

有时，自动检测依赖于一些设备拥有的默认特性。既然如此，驱动程序可以假定设备使用了这些默认值。这也是*short*在检测并口时的默认行为，正如*short*的代码所给出的那样，其实现相当简单：

```
if (short_irq < 0) /* 尚未指定: 强制使用默认值 */
    switch(short_base) {
        case 0x378: short_irq = 7; break;
        case 0x278: short_irq = 2; break;
        case 0x3bc: short_irq = 5; break;
    }
```

这段代码根据选定的I/O地址的基地址分配中断号，也允许用户在装载时用下面的命令行来覆盖默认值：

```
insmod ./short.ko irq=x
```

`short_base`默认为`0x378`，所以`short_irq`默认为`7`。

有些设备的设计更为先进，会简单地“声明”它们要使用的中断。这样，驱动程序就可以通过从设备的某个I/O端口或者PCI配置空间中读出一个状态字来获得中断号。当目标设备有能力告诉驱动程序它将使用的中断号时，自动检测IRQ号只是意味着探测设备，而不需要额外的工作来探测中断。幸运的是，大多数现代硬件以这种方式工作，比如，PCI标准要求外设声明它们打算使用的中断线，这样就能解决这个问题。我们将在第十二章深入讨论PCI标准。

令人遗憾的是，并不是所有的设备都对程序员很友好，自动检测可能还是需要做一些探测工作。这在技术上很简单：驱动程序通知设备产生中断并观察会发生什么。如果一切正常，那么只有一条中断信号线被激活。

尽管从理论上讲探测过程很简单，但实际上实现起来可就不那么清晰了。我们看看执行该任务的两种方法：调用内核定义的辅助函数，或者实现我们自己的版本。

内核帮助下的探测

Linux内核提供了一个底层设施来探测中断号。它只能在非共享中断的模式下工作，但是大多数硬件有能力工作在共享中断的模式下，并可提供更好的找到配置中断号的方法。内核提供的这一设施由两个函数组成，在头文件<*linux/interrupt.h*>中声明（该文件也描述了探测机制）：

`unsigned long probe_irq_on(void);`

这个函数返回一个未分配中断的位掩码。驱动程序必须保存返回的位掩码，并且将它传递给后面的*probe_irq_off*函数，调用该函数之后，驱动程序要安排设备产生至少一次中断。

`int probe_irq_off(unsigned long);`

在请求设备产生中断之后，驱动程序调用这个函数，并将前面*probe_irq_on*返回的位掩码作为参数传递给它。*probe_irq_off*返回“probe_irq_on”之后发生的中断编号。如果没有中断发生，就返回0（因此，IRQ 0不能被探测到，但在任何已支持的体系结构上，没有任何设备能够使用IRQ 0）。如果产生了多次中断（出现二义性），*probe_irq_off*会返回一个负值。

程序员要注意在调用*probe_irq_on*之后启用设备上的中断，并在调用*probe_irq_off*之前禁用中断。此外要记住，在*probe_irq_off*之后，需要处理设备上待处理的中断。

*short*模块演示了如何进行这样的探测。如果指定`probe=1`选项装载模块，并且并口连接器的引脚9和10相连，就会执行下面的代码进行中断信号线的检测：

```
int count = 0;
do {
    unsigned long mask;

    mask = probe_irq_on();
    outb_p(0x10,short_base+2); /* 启用中断报告 */
    outb_p(0x00,short_base);   /* 清除该位 */
    outb_p(0xFF,short_base);   /* 设置该位: 中断! */
    outb_p(0x00,short_base+2); /* 禁用中断报告 */
    udelay(5);  /* 留给中断探测一些时间 */
    short_irq = probe_irq_off(mask);

    if (short_irq =  = 0) { /* 没有找到? */
        printk(KERN_INFO "short: no irq reported by probe\n");
        short_irq = -1;
    }
    /*
     * 如果已经有多个中断线被激活，则结果为负值。
     * 我们应该服务该中断（lpt端口并不需要）并再次重试。
     * 最多重试五次，然后放弃。
     */
```

```
    } while (short_irq < 0 && count++ < 5);
    if (short_irq < 0)
        printk("short: probe failed %i times, giving up\n", count);
```

注意在调用*probe_irq_off*之前对*udelay*的使用。这取决于所使用的处理器速度，读者可能不得不安排一个很短的延时，以保证留给中断足够的传递时间。

探测是一个很耗时的任务，尽管*short*的探测耗时不多，但是像帧捕捉卡的探测就至少需要20毫秒的延迟（这对处理器来说已经是很长的时间了），而探测其他的设备可能要花费更多的时间。因此，最好的方法就是只在模块初始化的时候探测中断信号线一次，这与是否在设备打开时（应该这样做）或者在初始化函数内（不推荐这样做）安装中断处理例程无关。

值得注意的是，在一些平台（PowerPC、M68k、大部分MIPS的实现以及两个SPARC版本）上，探测是没有必要的，因此前面的函数只是一些空的占位符，有时叫做“useless ISA nonsense”。而在其他的平台上探测只是为ISA设备实现的。总之，大多数体系结构都定义了函数（甚至是空的）来简化现有的设备驱动程序的移植。

DIY探测

探测也可以由驱动程序自己实现。如果装载时指定`probe=2`，*short*模块将对IRQ信号线进行DIY探测。

这种机制与先前描述的内核帮助下的探测是一样的：启用所有未被占用的中断，然后观察会发生什么。但是，我们要充分发挥对有关设备的了解。通常，设备可以使用3或4个IRQ号中的一个来进行配置，探测这些IRQ号，使我们能够不必测试所有可能的IRQ就检测到正确的IRQ号。

在*short*的实现中，我们假定可能的IRQ值是3、5、7和9，这些编号实际上是并口设备允许用户选择的一些中断编号值。

下面的代码通过测试所有“可能”的中断并观察将要发生的事情来进行中断探测。`trials`数组列出了以0作为结束标志的需要测试的IRQ，`tried`数组用来记录哪个处理例程被驱动程序注册了。

```
int trials[] = {3, 5, 7, 9, 0};
int tried[]  = {0, 0, 0, 0, 0};
int i, count = 0;

/*
 * 为所有可能的中断线安装探测处理例程。
 * 记录那些目前空闲的结果（0为成功，否则为-EBUSY）仅仅是为了释放已获得的资源
 */
```

```
for (i = 0; trials[i]; i++)
    tried[i] = request_irq(trials[i], short_probing,
            SA_INTERRUPT, "short probe", NULL);

do {
    short_irq = 0; /* 什么也未获得 */
    outb_p(0x10,short_base+2); /* 启用中断 */
    outb_p(0x00,short_base);
    outb_p(0xFF,short_base); /* 切换中断位 */
    outb_p(0x00,short_base+2); /* 禁用中断 */
    udelay(5);  /* 留给探测一些时间 */

    /* 处理例程已经设置了相应的值 */
    if (short_irq =  = 0) { /* 没有找到? */
        printk(KERN_INFO "short: no irq reported by probe\n");
    }
    /*
     * 如果已经有多个中断线被激活，则结果为负。
     * 我们应该服务该中断（lpt端口并不需要）并再次重试。
     * 最多重试五次。
     */
} while (short_irq <=0 && count++ < 5);

/* 在循环结束后，卸载处理例程 */
for (i = 0; trials[i]; i++)
    if (tried[i] =  = 0)
        free_irq(trials[i], NULL);

if (short_irq < 0)
    printk("short: probe failed %i times, giving up\n", count);
```

有时，我们无法预知“可能的”IRQ值。在这种情况下，需要探测所有的空闲中断号，而不仅仅是那些由trials[]数组列出的中断号。为了探测所有的中断，不得不从IRQ 0探测到IRQ NR_IRQS-1，NR_IRQS是在头文件*<asm/irq.h>*中定义的具有平台相关性的常数。

现在我们就剩下探测处理例程本身了，处理例程的任务是根据实际收到的中断号更新short_irq变量，short_irq的值为0意味着“什么也没有”，负值意味着存在“二义性”。我们选择这些值是为了和*probe_irq_off*保持一致，这样，就可以在*short.c*中使用同样的代码调用任何一种探测方法。

```
irqreturn_t short_probing(int irq, void *dev_id, struct pt_regs *regs)
{
    if (short_irq =  = 0) short_irq = irq;    /* 找到 */
    if (short_irq != irq) short_irq = -irq; /* 出现二义性 */
    return IRQ_HANDLED;
}
```

处理例程的参数将在稍后介绍。只要了解参数irq是要处理的中断号，就足以理解上面的函数了。

快速和慢速处理例程

老版本的Linux内核做了很多努力才区分出“快速”和“慢速”中断。快速中断是那些可以很快被处理的中断，而处理慢速中断则会明显花费更长的时间。当慢速中断正被处理时，慢速中断要求处理器可以再次启用中断。否则，需要快速处理的任务可能会被延迟过长。

在现代内核中，很多快速中断和慢速中断的区别已经消失了。剩下的只有一个：快速中断（使用SA_INTERRUPT标志申请的中断）执行时，当前处理器上的其他所有中断都被禁止。注意，其他的处理器仍然可以处理中断，尽管从来不会看到两个处理器同时处理同一IRQ的情况。

那么，读者的驱动程序应该使用哪种中断处理例程呢？在现代系统中，SA_INTERRUPT只是在少数几种特殊情况（例如定时器中断）下使用。读者不应该使用SA_INTERRUPT标志，除非有足够必要的理由想要在其他中断被禁用的时候运行自己的中断处理例程。

这段论述足以满足大多数读者，但有些熟悉硬件或者对计算机有着强烈兴趣的读者或许需要深入了解一些信息。如果不想了解内部细节，可以跳过下一节。

x86平台上中断处理的内幕

下面的描述是从2.6内核中的文件*arch/i386/kernel/irq.c*、*arch/i386/kernel/apic.c*、*arch/i386/kernel/entry.S*、*arch/i386/kernel/i8259.c*以及*include/asm-i386/hw_irq.h*中得出的。虽然基本概念是相同的，但是硬件细节还是与其他平台有所区别。

最底层的中断处理代码可见*entry.S*文件，该文件是一个汇编语言文件，完成了许多机器级的工作。这个文件利用几个汇编技巧及一些宏，将一段代码用于所有可能的中断。在所有情况下，这段代码将中断编号压入栈，然后跳转到一个公共段，而这个公共段会调用在*irq.c*中定义的*do_IRQ*函数。

*do_IRQ*做的第一件事是应答中断，这样中断控制器就可以继续处理其他的事情了。然后该函数对于给定的IRQ号获得一个自旋锁，这样就阻止了任何其他的CPU处理这个IRQ。接着清除几个状态位（包括一个我们很快会讲到的IRQ_WAITING），然后寻找这个特定IRQ的处理例程。如果没有处理例程，就什么也不做；自旋锁被释放，处理任何待处理的软件中断，最后*do_IRQ*返回。

通常，如果设备有一个已注册的处理例程并且发生了中断，则函数*handle_IRQ_event*会被调用以便实际调用处理例程。如果处理例程是慢速类型（即SA_INTERRUPT未被设置），将重新启用硬件中断，并调用处理例程。然后只是做一些清理工作，接着运行软

件中断，最后返回到常规工作。作为中断的结果（例如，处理例程可以*wake_up*一个进程），“常规工作”可能已经被改变，所以，从中断返回时发生的最后一件事情可能就是一次处理器的重新调度。

IRQ的探测是通过为每个缺少中断处理例程的IRQ设置IRQ_WAITING状态位来完成的。当中断产生时，因为没有注册处理例程，*do_IRQ*清除该位然后返回。当*probe_irq_off*被一个驱动程序调用的时候，只需要搜索那些没有设置IRQ_WAITING位的IRQ。

实现中断处理例程

迄今为止，我们已经学会了如何注册一个中断处理例程，但是还没有编写过中断处理例程。实际上，处理例程没有什么与众不同的地方——它们也是普通的C程序。

唯一特殊的地方就是处理例程是在中断时间内运行的，因此它的行为会受到一些限制。这些限制与我们在内核定时器中看到的一样。处理例程不能向用户空间发送或者接收数据，因为它不是在任何进程的上下文中执行的，处理例程也不能做任何可能发生休眠的操作，例如调用*wait_event*、使用不带GFP_ATOMIC标志的内存分配操作，或者锁住一个信号量等等。最后，处理例程不能调用*schdule*函数。

中断处理例程的功能就是将有关中断接收的信息反馈给设备，并根据正在服务的中断的不同含义对数据进行相应的读或写。第一步通常要清除接口卡上的一个位，大多数硬件设备在它们的“interrupt-pending（中断挂起）”位被清除之前不会产生其他的中断。根据硬件的工作方式，这个步骤可能在最后而不是第一个步骤执行；也就是说，这里不存在永恒规则。有些设备不需要这个步骤，因为它们没有“中断挂起”位，这样的设备是很少的，但并口设备却是其中的一种。由于这个原因，*short*不需要清除这样的位。

中断处理例程的一个典型任务就是：如果中断通知进程所等待的事件已经发生，比如新的数据到达，就会唤醒在该设备上休眠的进程。

还是举帧捕捉卡的例子，一个进程通过连续地读该设备来获取一系列图像；在读每一帧数据前，*read*调用都是阻塞的，每当新的数据帧到达时，中断处理例程就会唤醒此进程。这里假定捕捉卡会中断处理器以便通知每一帧数据的成功到达。

无论是快速还是慢速处理例程，程序员都应该编写执行时间尽可能短的处理例程。如果需要执行一个长时间的计算任务，最好的方法是使用tasklet或者工作队列在更安全的时间内调度计算任务（我们将在“顶半和底半”一节中描述如何用这种方法来延迟处理工作）。

在*short*示例代码中，中断处理例程调用了*do_gettimeofday*，并输出当前时间到大小为

一页的循环缓冲区中，然后唤醒任何一个读取进程，告诉该进程现在有新的数据可以读取。

```
irqreturn_t short_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    struct timeval tv;
    int written;

    do_gettimeofday(&tv);

    /* 写入一个16字节的记录。假定 PAGE_SIZE 是 16 的倍数 */
    written = sprintf((char *)short_head,"%08u.%06u\n",
            (int)(tv.tv_sec % 100000000), (int)(tv.tv_usec));
    BUG_ON(written != 16);
    short_incr_bp(&short_head, written);
    wake_up_interruptible(&short_queue); /* 唤醒任何读取进程 */
    return IRQ_HANDLED;
}
```

尽管上述代码很简单，却代表了中断处理例程的典型工作流程。它所调用的*short_incr_bp*函数定义如下：

```
static inline void short_incr_bp(volatile unsigned long *index, int delta)
{
    unsigned long new = *index + delta;
    barrier();  /* 禁止对前后两条语句的优化 */
    *index = (new >= (short_buffer + PAGE_SIZE)) ? short_buffer : new;
}
```

这个函数的实现非常谨慎，它可以将指针限制在循环缓冲区的范围之内，并且不会因为传递一个不正确的值而返回。对*barrier*的调用将阻止编译器在函数的两行语句之间做任何优化工作。如果没有这个屏障，编译器可能会优化出一个新的变量并将其直接赋值给*index。在index发生反转的短暂时间内，这种优化可能会产生不正确的索引值。通过仔细处理其他线程可见的值来避免出现不一致的情况，我们就可以不使用锁而安全操作循环缓冲区的指针。

用来读取我们在中断时间内填充的缓冲区内容的设备文件是*/dev/shortint*。我们在第九章里并没有介绍这个设备特殊文件以及*/dev/shortprint*，因为它们的用法只是针对中断处理的。*/dev/shortint*的内部实现是专门针对中断的产生和报告的。每向设备写一个字节就产生一次中断，而读取设备时则给出每次中断报告产生的时间。

如果读者连接并口连接器的引脚9和10，通过拉高并口数据字节的高位就会产生中断，这可以通过向设备文件*/dev/short0*写入二进制数据或者向设备文件*/dev/shortint*写入任何数据来实现（注2）。

注2：*Shortint*设备通过交替地向并口写入0x00和0xff来完成其任务。

下面的代码实现了对 */dev/shortint* 的读取和写入：

```
ssize_t short_i_read (struct file *filp, char _ _user *buf, size_t count,
     loff_t *f_pos)
{
    int count0;
    DEFINE_WAIT(wait);

    while (short_head =  = short_tail) {
        prepare_to_wait(&short_queue, &wait, TASK_INTERRUPTIBLE);
        if (short_head =  = short_tail)
            schedule();
        finish_wait(&short_queue, &wait);
        if (signal_pending (current))  /* 某个信号已到达 */
            return -ERESTARTSYS; /* 告诉 fs 层来做进一步处理 */
    }
    /* count0 是可读取数据的字节数 */
    count0 = short_head - short_tail;
    if (count0 < 0) /* 已交换 */
        count0 = short_buffer + PAGE_SIZE - short_tail;
    if (count0 < count) count = count0;

    if (copy_to_user(buf, (char *)short_tail, count))
        return -EFAULT;
    short_incr_bp (&short_tail, count);
    return count;
}

ssize_t short_i_write (struct file *filp, const char _ _user *buf, size_t
count,
        loff_t *f_pos)
{
    int written = 0, odd = *f_pos & 1;
    unsigned long port = short_base; /* 输出到并口的数据锁存器 */
    void *address = (void *) short_base;

    if (use_mem) {
        while (written < count)
            iowrite8(0xff * ((++written + odd) & 1), address);
    } else {
        while (written < count)
            outb(0xff * ((++written + odd) & 1), port);
    }

    *f_pos += count;
    return written;
}
```

其他的设备特殊文件，如 */dev/shortprint*，使用并口来驱动一台打印机，如果读者想避免在D-25连接器的引脚9和10之间焊接一根电线的话，就可以使用打印机。*shortprint* 的 *write* 实现使用了一个循环缓冲区来存储被打印的数据，而 *read* 的实现是刚才介绍的那一种（所以读者可以读出打印机获得每个字符的时间）。

为了支持打印机操作，上面列出的中断处理例程被做了少许修改，增加了发送下一个数据字节到打印机的能力（如果有更多的数据需要传送的话）。

处理例程的参数及返回值

虽然 *short* 没有对参数进行处理，但还是有三个参数被传给了中断处理例程：irq、dev_id和regs。让我们看看每个参数的意义。

如果存在任何可以打印到日志的消息时，中断号（int irq）是很有用的。第二个参数void *dev_id是一种客户数据类型（即驱动程序可用的私有数据）。传递给*request_irq*函数的void *参数会在中断发生时作为参数被传回处理例程。通常，我们会为dev_id传递一个指向自己设备的数据结构指针，这样，一个管理若干同样设备的驱动程序在中断处理例程中不需要做任何额外的代码，就可以找出哪个设备产生了当前的中断事件。

中断处理例程中参数的典型用法如下：

```
static irqreturn_t sample_interrupt(int irq, void *dev_id, struct pt_regs
                                    *regs)
{
    struct sample_dev *dev = dev_id;

    /* 现在，dev指向正确的硬件项 */
    /* .... */
}
```

与这个处理例程相关联的典型 *open* 代码如下所示：

```
static void sample_open(struct inode *inode, struct file *filp)
{
    struct sample_dev *dev = hwinfo + MINOR(inode->i_rdev);
    request_irq(dev->irq, sample_interrupt,
                0 /* flags */, "sample", dev /* dev_id */);
    /*....*/
    return 0;
}
```

最后一个参数struct pt_reg *regs很少使用，它保存了处理器进入中断代码之前的处理器上下文快照。该寄存器可被用来监视和调试，对一般的设备驱动程序任务来说通常不是必需的。

中断处理例程应该返回一个值，用来指明是否真正处理了一个中断。如果处理例程发现其设备的确需要处理，则应该返回 IRQ_HANDLED；否则，返回值应该是 IRQ_NONE。我们也可以通过下面的宏来产生这个返回值：

```
IRQ_RETVAL(handled)
```

如果要处理该中断，则handled应该取非零值。该返回值将被内核使用，以便检测并抑制假的中断。如果设备无法告诉我们是否被真正中断，则应该返回 IRQ_HANDLED。

启用和禁用中断

有时设备驱动程序必须在一个时间段内（希望较短）阻塞中断的发出（我们在第五章的“自旋锁”一节中看到过这种情况）。通常来说，我们必须在拥有自旋锁的时候阻塞中断，以免死锁系统。在涉及自旋锁的情况下，有多种办法可禁用中断。但在我们讨论这些办法之前，要注意应该尽量少禁用中断，即使在设备驱动程序中也是如此，同时这种技术不应在驱动程序中作为互斥机制使用。

禁用单个中断

有时（但很少），驱动程序需要禁用某个特定中断线的中断产生。为此，内核提供了三个函数，这些函数均在 *<asm/irq.h>* 中声明。这些函数是内核 API 的一部分，因此在这里描述它们，但对大多数驱动程序来讲，我们并不鼓励使用这些函数。此外，我们不能禁用共享的中断线，而在现代系统上，中断的共享是很常见的。这三个函数的原型如下：

```
void disable_irq(int irq);
void disable_irq_nosync(int irq);
void enable_irq(int irq);
```

调用这些函数中的任何一个都会更新可编程中断控制器（Programmable Interrupt Controller，PIC）中指定中断的掩码，因而就可以在所有的处理器上禁用或者启用IRQ。对这些函数的调用是可以嵌套的 —— 如果 *disable_irq* 被成功调用两次，那么在IRQ真正重新启用之前，则需要执行两次 *enable_irq* 调用。从一个中断处理例程中调用这些函数是可以的，但是在处理某个 IRQ 时再打开它并不是一个好习惯。

disable_irq 不但会禁止给定的中断，而且也会等待当前正在执行的中断处理例程完成。要明白的是，如果调用 *disable_irq* 的线程拥有任何中断处理例程需要的资源（比如自旋锁），则系统会死锁。和 *disable_irq* 不同，*disable_irq_nosync* 是立即返回的。因此使用后者将会更快，但是可能会让你的驱动程序处于竞态状态。

但为什么还要禁用中断呢？还是举并口的例子，先看看 *plip* 网络接口。一个 *plip* 设备使用裸的并口传送数据。因为并口连接器上只有 5 个位可以读，它们被解释为 4 个数据位和一个时钟/握手信号。当发起者（发送数据包的那个接口）送出一个包的头 4 个位时，时钟线的电平升高，这将导致接收方接口中断处理器。然后，*plip* 的处理例程就会被调用，以便处理最新到达的数据。

在设备被通知之后，数据的传输将继续进行。这里，*plip* 使用握手信号线和接收方保持同步（这可能不是最好的实现，但是和其他使用并口的数据包驱动程序保持兼容是必要的）。如果接收接口每接收一个字节都要处理两次中断，那么性能显然是不可忍受的。因此驱动程序在接收数据包的时候禁用中断，而使用“轮询并延迟”循环来接收数据。

同样地，因为接收方到发送方的握手信号被用来应答数据的接收，所以发送接口也要在发送数据包时禁用它自己的IRQ信号。

禁用所有的中断

如果要禁用所有的中断该怎么办？在2.6内核中，可通过下面两个函数之一关闭当前处理器上的所有中断处理，这两个函数定义在 *<asm/system.h>* 中：

```
void local_irq_save(unsigned long flags);
void local_irq_disable(void);
```

对 *local_irq_save* 的调用将把当前中断状态保存到 `flags` 中，然后禁用当前处理器上的中断发送。注意，`flags` 被直接传递，而不是通过指针来传递。*local_irq_disable* 不保存状态而关闭本地处理器上的中断发送；只有我们知道中断并未在其他地方被禁用的情况下，才能使用这个版本。

可通过如下函数打开中断：

```
void local_irq_restore(unsigned long flags);
void local_irq_enable(void);
```

第一个版本会将 *local_irq_save* 保存的 `flags` 状态值恢复，而 *local_irq_enable* 无条件打开中断。与 *disable_irq* 不同，*local_irq_disable* 不会维护对多次的调用的跟踪。如果调用链中的多个函数需要禁用中断，则应该使用 *local_irq_save*。

在2.6内核中，没有办法全局禁用整个系统上的所有中断。内核开发者认为关闭所有中断的代价太高，因此没有必要提供这种能力。如果读者使用的老驱动程序调用了类似 *cli* 和 *sti* 这样的函数，为了该驱动程序能够在2.6下使用，则需要进行修改而使用正确的锁。

顶半部和底半部

中断处理的一个主要问题是怎样在处理例程内完成耗时的任务。响应一次设备中断需要完成一定数量的工作，但是中断处理例程需要尽快结束而不能使中断阻塞的时间过长，这两个需求（工作和速度）彼此冲突，让驱动程序的作者多少有点困扰。

Linux（连同很多其他的系统）通过将中断处理例程分成两部分来解决这个问题。称为“顶半部”的部分，是实际响应中断的例程，也就是用*request_irq*注册的中断例程；而所谓的“底半部”是一个被顶半部调度，并在稍后更安全的时间内执行的例程。顶半部处理例程和底半部处理例程之间最大的不同，就是当底半部处理例程执行时，所有的中断都是打开的——这就是所谓的在更安全时间内运行。典型的情况是顶半部保存设备的数据到一个设备特定的缓冲区并调度它的底半部，然后退出：这个操作是非常快的。然后，底半部执行其他必要的工作，例如唤醒进程、启动另外的I/O操作等等。这种方式允许在底半部工作期间，顶半部还可以继续为新的中断服务。

几乎每一个严格的中断处理例程都是以这种方式分成两部分的。例如，当一个网络接口报告有新数据包到达时，处理例程仅仅接收数据并将它推到协议层上，而实际的数据包处理过程是在底半部执行的。

Linux内核有两种不同的机制可以用来实现底半部处理，我们已经在第七章介绍过这两种机制了。tasklet通常是底半部处理的优选机制；因为这种机制非常快，但是所有的tasklet代码必须是原子的。除了tasklet之外，我们还可以选择工作队列，它可以具有更高的延迟，但允许休眠。

下面再次用*short*驱动程序来进行我们的讨论。在使用某个模块选项装载时，可以通知*short*模块使用顶/底半部的模式进行中断处理，并采用tasklet或者工作队列处理例程。因此，顶半部执行得就很快，因为它仅保存当前时间并调度底半部处理。然后底半部负责这些时间的编码，并唤醒可能等待数据的任何用户进程。

tasklet

记住tasklet是一个可以在由系统决定的安全时刻在软件中断上下文被调度运行的特殊函数。它们可以被多次调度运行，但tasklet的调度并不会累积；也就是说，实际只会运行一次，即使在激活tasklet的运行之前重复请求该tasklet的运行也是这样。不会有同一tasklet的多个实例并行地运行，因为它们只运行一次，但是tasklet可以与其他的tasklet并行地运行在对称多处理器（SMP）系统上。这样，如果驱动程序有多个tasklet，它们必须使用某种锁机制来避免彼此间的冲突。

tasklet可确保和第一次调度它们的函数运行在同样的CPU上。这样，因为tasklet在中断处理例程结束前并不会开始运行，所以此时的中断处理例程是安全的。不管怎样，在tasklet运行时，当然可以有其他的中断发生，因此在tasklet和中断处理例程之间的锁还是需要的。

必须使用宏DECLARE_TASKLET声明tasklet:

```
DECLARE_TASKLET(name, function, data);
```

name是给tasklet起的名字，*function*是执行tasklet时调用的函数(它带有一个unsigned long型的参数并且返回void)，data是一个用来传递给*tasklet*函数的unsigned long类型的值。

驱动程序*short*如下声明它自己的tasklet:

```
void short_do_tasklet(unsigned long);
DECLARE_TASKLET(short_tasklet, short_do_tasklet, 0);
```

函数*tasklet_schedule*用来调度一个tasklet运行。如果指定tasklet=1选项装载*short*，它就会安装一个不同的中断处理例程，这个处理例程保存数据并如下调度tasklet:

```
irqreturn_t short_tl_interrupt(int irq, void *dev_id, struct pt_regs
*regs)
{
    do_gettimeofday((struct timeval *) tv_head); /* 强制转换以免出现“易失性”
警告 */
    short_incr_tv(&tv_head);
    tasklet_schedule(&short_tasklet);
    short_wq_count++; /* 记录中断的产生 */
    return IRQ_HANDLED;
}
```

实际的tasklet例程，即*short_do_tasklet*，将会在系统方便时得到执行。就像先前提到的，这个例程执行中断处理的大多数任务，如下所示:

```
void short_do_tasklet (unsigned long unused)
{
    int savecount = short_wq_count, written;
    short_wq_count = 0; /* 已经从队列中移除 */
    /*
     * 底半部读取由顶半部填充的tv数组，
     * 并向循环文本缓冲区中打印信息，而缓冲区的数据则由
     * 读取进程获得
     */

    /* 首先将调用此bh之前发生的中断数量写入 */
    written = sprintf((char *)short_head,"bh after %6i\n",savecount);
    short_incr_bp(&short_head, written);

    /*
     * 然后写入时间值。每次写入16字节，
     * 所以它与 PAGE_SIZE是对齐的
     */

    do {
        written = sprintf((char *)short_head,"%08u.%06u\n",
                (int)(tv_tail->tv_sec % 100000000),
                (int)(tv_tail->tv_usec));
```

```
            short_incr_bp(&short_head, written);
            short_incr_tv(&tv_tail);
        } while (tv_tail != tv_head);
        wake_up_interruptible(&short_queue); /* 唤醒任何读取进程 */
    }
```

在其他动作之外，这个tasklet记录了自从它上次被调用以来产生了多少次中断。一个类似于*short*的设备可以在很短的时间内产生很多次中断，所以在底半部被执行前，肯定会有多次中断发生。驱动程序必须一直对这种情况有所准备，并且必须能根据顶半部保留的信息知道有多少工作需要完成。

工作队列

读者应该记得，工作队列会在将来的某个时间、在某个特殊的工作者进程上下文中调用一个函数。因为工作队列函数运行在进程上下文中，因此可在必要时休眠。但是我们不能从工作队列向用户空间复制数据，除非使用将在第十五章中描述的高级技术，要知道，工作者进程无法访问其他任何进程的地址空间。

如果在装载*short*驱动程序时将wq选项设置为非零值，则该驱动程序将使用工作队列作为其底半部进程。它使用系统的默认工作队列，因此不需要其他特殊的设置代码；但是，如果我们的驱动程序具有特殊的延迟需求（或者可能在工作队列函数中长时间休眠），则应该创建我们自己的专用工作队列。我们需要一个work_struct结构，该结构如下声明并初始化：

```
static struct work_struct short_wq;

    /* 下面这行出现在short_init()中 */
    INIT_WORK(&short_wq, (void (*)(void *)) short_do_tasklet, NULL);
```

我们的工作者函数是*short_do_tasklet*，该函数已经在先前的小节中介绍过了。

在使用工作队列时，*short*构造了另一个中断处理例程，如下所示：

```
irqreturn_t short_wq_interrupt(int irq, void *dev_id, struct pt_regs
*regs)
{
    /* 获取当前的时间信息。*/
    do_gettimeofday((struct timeval *) tv_head);
    short_incr_tv(&tv_head);

    /* 排序bh。不必关心多次调度的情况 */
    schedule_work(&short_wq);

    short_wq_count++; /* 记录中断的到达 */
    return IRQ_HANDLED;
}
```

读者可以看到，该中断处理例程和tasklet版本非常相似，唯一的不同是它调用*schedule_work*来安排底半部处理。

中断共享

“IRQ冲突”这种说法和“PC架构”几乎是同义语。通常，PC上的IRQ信号线不能为一个以上的设备服务，它们从来都是不够用的，结果，许多没有经验的用户总是花费很多时间试图找到一种方法使所有的硬件能够协同工作，因此他们不得不总是打开自己计算机的外壳。

当然，现代硬件已经能允许中断的共享了，比如PCI总线就要求外设可共享中断。因此，Linux内核支持所有总线的中断共享，即使在类似ISA这样原先并不支持共享的总线上。针对2.6内核的设备驱动程序，应该在目标硬件可以支持共享中断操作的情况下处理中断的共享。幸运的是，大多数情况下很容易使用共享的中断。

安装共享的处理例程

就像普通非共享的中断一样，共享的中断也是通过*request_irq*安装的，但是有两处不同：

- 请求中断时，必须指定flags参数中的SA_SHIRQ位。
- dev_id参数必须是唯一的。任何指向模块地址空间的指针都可以使用，但dev_id不能设置成NULL。

内核为每个中断维护了一个共享处理例程的列表，这些处理例程的dev_id各不相同，就像是设备的签名。如果两个驱动程序在同一个中断上都注册NULL作为它们的签名，那么在卸载的时候引起混淆，当中断到达时造成内核出现oops消息。由于这个原因，在注册共享中断时如果传递了值为NULL的dev_id，现代的内核就会给出警告。当请求一个共享中断时，如果满足下面条件之一，那么*request_irq*就会成功：

- 中断信号线空闲。
- 任何已经注册了该中断信号线的处理例程也标识了IRQ是共享的。

无论何时，当两个或者更多的驱动程序共享同一根中断信号线，而硬件又通过这根信号线中断处理器时，内核会调用每一个为这个中断注册的处理例程，并将它们自己的dev_id传回去。因此，一个共享的处理例程必须能够识别属于自己的中断，并且在自己的设备没有被中断的时候迅速退出。

如果读者在请求中断请求信号线之前需要探测设备的话，则内核不会有所帮助，对于共

享的处理例程是没有探测函数可以利用的。仅当要使用的中断信号线处于空闲时，标准的探测机制才能工作。但如果信号线已经被其他具有共享特性的驱动程序占用的话，即使你的驱动已经可以很好的工作了，探测也会失败。幸运的是，多数可共享中断的硬件能够告诉处理器它们在使用哪个中断，这样就消除了显式探测的必要。

释放处理例程同样是通过执行*free_irq*来实现的。这里dev_id参数被用来从该中断的共享处理例程列表中选择正确的处理例程来释放，这就是为什么dev_id指针必须唯一的原因。

使用共享处理例程的驱动程序需要小心一件事情：不能使用*enable_irq*和*disable_irq*。如果使用了，共享中断信号线的其他设备就无法正常工作了；即使在很短的时间内禁用中断，也会因为这种延迟而为设备和其用户带来问题。通常，程序员必须记住他的驱动程序并不独占IRQ，所以它的行为必须比独占中断信号线时更“社会化”。

运行处理例程

如上所述，当内核收到中断时，所有已注册的处理例程都将被调用。一个共享中断处理例程必须能够将要处理的中断和其他设备产生的中断区分开来。

装载*short*时，如果指定shared=1选项，则将安装下面的处理例程而不是默认的处理例程：

```
irqreturn_t short_sh_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    int value, written;
    struct timeval tv;

    /* 如果不是short产生的，则立即返回 */
    value = inb(short_base);
    if (!(value & 0x80))
        return IRQ_NONE;

    /* 清除中断位 */
    outb(value & 0x7F, short_base);

    /* 其余部分没有什么变化 */

    do_gettimeofday(&tv);
    written = sprintf((char *)short_head,"%08u.%06u\n",
            (int)(tv.tv_sec % 100000000), (int)(tv.tv_usec));
    short_incr_bp(&short_head, written);
    wake_up_interruptible(&short_queue); /* 唤醒任何的读取进程 */
    return IRQ_HANDLED;
}
```

解释如下。因为并口没有“interrupt-pending”位可以检查，为此处理例程使用了ACK位。如果该位为高，那么报告的中断就是送给*short*的，然后处理例程清除该位。

处理例程通过将并口的数据端口的高位清零来重新设置该位——*short*假定并口的引脚9和10是连接在一起的。如果一个与*short*共享IRQ的其他设备产生了中断，*short*就知道它自己的信号线没有被激活，所以不会做任何工作。

一个功能完整的驱动程序可能会将任务分成顶半部和底半部，当然这很容易添加，并且对用于实现共享的代码没有任何影响。一个真正的驱动程序或许会使用dev_id参数来判断产生中断的某个或多个设备。

注意，如果读者使用一台打印机（代替跳线）来检验*short*的中断管理，那么这个共享的中断处理例程不会按预期工作，因为打印机协议不允许共享，而且驱动程序也无从知道中断是否是由打印机产生的。

/proc接口和共享的中断

在系统上安装共享的中断处理例程不会对*/proc/stat*造成影响，它甚至不知道哪些处理例程是共享的，但是，*/proc/interrupts*会有稍许改变。

所有为同一个中断号安装的处理例程会出现在*/proc/interrupts*文件的同一行上。下面的输出（来自一个x86_64系统）说明了共享的中断处理例程是怎样显示的：

```
           CPU0
  0:  892335412     XT-PIC  timer
  1:     453971     XT-PIC  i8042
  2:          0     XT-PIC  cascade
  5:          0     XT-PIC  libata, ehci_hcd
  8:          0     XT-PIC  rtc
  9:          0     XT-PIC  acpi
 10:   11365067     XT-PIC  ide2, uhci_hcd, uhci_hcd, SysKonnect SK-98xx, EMU10K1
 11:    4391962     XT-PIC  uhci_hcd, uhci_hcd
 12:        224     XT-PIC  i8042
 14:    2787721     XT-PIC  ide0
 15:     203048     XT-PIC  ide1
NMI:      41234
LOC:  892193503
ERR:        102
MIS:          0
```

该系统具有多个共享的中断线。IRQ 5用于串行ATA和IEEE 1394控制器；IRQ 10也由多个设备共享，包括一个IDE控制器、两个USB控制器、一个以太网接口以及一个声卡；IRQ 11也由两个USB控制器使用。

中断驱动的I/O

如果与驱动程序管理的硬件之间的数据传输因为某种原因被延迟的话，驱动程序作者就应该实现缓冲。数据缓冲区有助于将数据的传送和接收与系统调用*write*和*read*分离开来，从而提高系统的整体性能。

一个好的缓冲机制需要采用中断驱动的I/O，这种模式下，一个输入缓冲区在中断时间内被填充，并由读取该设备的进程取走缓冲区内的数据；一个输出缓冲区由写入设备的进程填充，并在中断时间内取走数据。一个中断驱动输出的例子是*/dev/shortint*的实现。

要正确进行中断驱动的数据传输，则要求硬件应该能按照下面的语义来产生中断：

- 对于输入来说，当新的数据已经到达并且处理器准备好接收它时，设备就中断处理器。实际执行的动作取决于设备使用的是I/O端口、内存映射，还是DMA。
- 对于输出来说，当设备准备好接收新数据或者对成功的数据传送进行应答时，就要发出中断。内存映射和具有DMA能力的设备，通常通过产生中断来通知系统它们对缓冲区的处理已经结束。

*read*或者*write*与实际数据到达之间的时序关系已经在第六章“阻塞和非阻塞式操作”一节中介绍过。

写缓冲区示例

我们已多次提到*shortprint*驱动程序，现在可以看看实际的代码了。这个模块实现了一个针对并口的、面向输出的非常简单的驱动程序；但是，该驱动程序足够用来打印文件了。当然，在测试这个驱动程序的打印功能时，记住要以打印机可理解的文件格式发送，要知道，并不是所有的打印机都能对任意数据的流给出相应的响应。

*shortprint*驱动程序维护了一页大小的输出用循环缓冲区。当用户空间进程向该设备写入数据时，数据会反馈到缓冲区中，但*write*方法并不实际执行任何的I/O操作。*shortp_write*的核心代码如下所示：

```
while (written < count) {
    /* 挂起直到有可用缓冲区空间为止。*/
    space = shortp_out_space();
    if (space <= 0) {
        if (wait_event_interruptible(shortp_out_queue,
                    (space = shortp_out_space()) > 0))
            goto out;
    }
```

```
            /* 将数据移动到缓冲区。*/
            if ((space + written) > count)
                space = count - written;
            if (copy_from_user((char *) shortp_out_head, buf, space)) {
                up(&shortp_out_sem);
                return -EFAULT;
            }
            shortp_incr_out_bp(&shortp_out_head, space);
            buf += space;
            written += space;

            /* 如果没有激活的输出，则激活。*/
            spin_lock_irqsave(&shortp_out_lock, flags);
            if (! shortp_output_active)
                shortp_start_output();
            spin_unlock_irqrestore(&shortp_out_lock, flags);
        }

    out:
        *f_pos += written;
```

信号量（shortp_out_sem）控制着对该循环缓冲区的访问；在上述代码之前，*shortp_write* 会获得该信号。在拥有该信号量的同时，它会尝试将数据反馈到循环缓冲区。函数 *shortp_out_space* 返回连续可用的空间大小（因此没有必要关心缓冲区的反转问题）；如果该大小为0，驱动程序就等待直到空间被释放。然后，它会将数据复制到该缓冲区。

一旦有数据要输出，*shortp_write* 必须确保数据被写入设备。实际的写入由工作队列函数完成；*shortp_write* 必须在该工作队列函数尚未执行时调度该函数。在获取一个独立的自旋锁（该自旋锁控制对用于输出缓冲区消费者端的变量的访问，其中包括shortp_output_active）之后，必要时它调用 *shortp_start_output*。之后，只需关注有多少数据“已写入”该缓冲区并返回。

启动输出进程的函数如下所示：

```
    static void shortp_start_output(void)
    {
        if (shortp_output_active) /* 不应发生 */
            return;

        /* 设置“丢失中断”定时器 */
        shortp_output_active = 1;
        shortp_timer.expires = jiffies + TIMEOUT;
        add_timer(&shortp_timer);

        /*  然后让进程继续。*/
        queue_work(shortp_workqueue, &shortp_work);
    }
```

处理硬件的实际代码有时会丢失来自设备的中断。如果发生这种情况，我们不希望驱动程序永久停止直到系统重启；也就是说，我们必须以用户友好的方式解决这个问题。如果能够意识到中断的丢失并继续处理则会好得多。为此，*shortprint*设置了一个内核定时器来向设备输出数据。如果该定时器到期，则可能丢失了某个中断。我们很快就能看到定时器函数，但是现在仍然关注主要的输出功能。这个功能在工作队列函数中实现，我们已经看到该函数被调度，而其核心代码如下所示：

```
spin_lock_irqsave(&shortp_out_lock, flags);

/* 是否有数据写入？ */
if (shortp_out_head = = shortp_out_tail) { /* 空的 */
    shortp_output_active = 0;
    wake_up_interruptible(&shortp_empty_queue);
    del_timer(&shortp_timer);
}
/* 否则写入其他字节 */
else
    shortp_do_write();

/* 如果有人等待，则唤醒之。 */
if (((PAGE_SIZE + shortp_out_tail - shortp_out_head) % PAGE_SIZE) > SP_MIN_SPACE)
{
    wake_up_interruptible(&shortp_out_queue);
}
spin_unlock_irqrestore(&shortp_out_lock, flags);
```

因为我们正在处理输出端的共享变量，因此必须获得自旋锁。然后，代码会检查是否存在数据需要发送；如果没有，我们记录该输出不再活动，删除定时器，并唤醒任何可能等待该队列彻底为空（在设备被关闭时将发生此类等待）的进程。相反，如果仍有数据要写入，则调用 *shortp_do_write* 来真正将一个字节发送给硬件。

然后，因为我们可能已经在输出缓冲区中有了空闲空间，因此需要考虑唤醒那些等待添加数据到该缓冲区的进程。但是我们并不是无条件地执行这个唤醒，相反，我们只会在空闲空间达到某个最小值时才会执行唤醒。每次缓冲区中的一个字节被发送给硬件之后就唤醒写入进程，并不是好的选择；因为唤醒一个进程，调度该进程运行，然后将其重新置于休眠的成本太高了。相反，我们应该等待，直到进程可以将具有实质性大小的数据放到缓冲区。这种技术在缓冲的、中断驱动的驱动程序中非常常见。

为了完整，下面给出将数据真正写入端口的代码：

```
static void shortp_do_write(void)
{
    unsigned char cr = inb(shortp_base + SP_CONTROL);

    /* 有情况发生，重置定时器 */
    mod_timer(&shortp_timer, jiffies + TIMEOUT);
```

```
    /* 向设备输出一个字节 */
    outb_p(*shortp_out_tail, shortp_base+SP_DATA);
    shortp_incr_out_bp(&shortp_out_tail, 1);
    if (shortp_delay)
        udelay(shortp_delay);
    outb_p(cr | SP_CR_STROBE, shortp_base+SP_CONTROL);
    if (shortp_delay)
        udelay(shortp_delay);
    outb_p(cr & ~SP_CR_STROBE, shortp_base+SP_CONTROL);
}
```

这里，我们重置定时器以反映我们已经完成了某些处理，然后发送一个字节到设备，最后更新循环缓冲区的指针。

工作队列函数不会直接提交自己，因此将只会有一个字节被写入设备。打印机将以自己的慢速方式处理这个字节，之后等待处理下一个字节；然后会中断处理器。*shortprint*使用的中断处理例程很短，也很简单：

```
static irqreturn_t shortp_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    if (! shortp_output_active)
        return IRQ_NONE;
    /* 记录时间，其他信息在工作队列函数中获得 */
    do_gettimeofday(&shortp_tv);
    queue_work(shortp_workqueue, &shortp_work);
    return IRQ_HANDLED;
}
```

因为并口并不需要显式的中断应答，因此中断处理例程唯一要做的就是告诉内核再次运行工作队列。

如果中断始终不产生会怎么样呢？我们已经看到的驱动程序代码会导致停顿。为了避免这种情况的发生，我们在前几页设置了一个定时器。该定时器到期时会执行下面的函数：

```
static void shortp_timeout(unsigned long unused)
{
    unsigned long flags;
    unsigned char status;

    if (! shortp_output_active)
        return;
    spin_lock_irqsave(&shortp_out_lock, flags);
    status = inb(shortp_base + SP_STATUS);

    /* 如果打印机仍然忙，则只是重置定时器 */
    if ((status & SP_SR_BUSY) == 0 || (status & SP_SR_ACK)) {
        shortp_timer.expires = jiffies + TIMEOUT;
        add_timer(&shortp_timer);
        spin_unlock_irqrestore(&shortp_out_lock, flags);
        return;
    }
```

```
        /* 否则必须调用中断处理例程 */
        spin_unlock_irqrestore(&shortp_out_lock, flags);
        shortp_interrupt(shortp_irq, NULL, NULL);
    }
```

如果没有激活的输出，则定时器函数会直接返回；这可避免在发生问题时定时器再次提交自身。然后，在获取自旋锁之后，我们查询端口的状态；如果端口正忙，则说明端口尚未发送中断给我们，因此我们重置定时器并返回。打印机有时可能会花费很长时间才能准备就绪，比如管理员正在休假时打印机缺纸的情况。在这种情况下，除了耐心等待情况发生变化之外别无他法。

但是，如果打印机声明已就绪，则说明我们丢失了它的中断。在这种情况下，我们只要简单地手工调用我们的中断处理例程就可让输出过程重新继续。

*shortprint*驱动程序不支持端口的读取；相反，它像*shortint*一样返回中断定间信息。但是，中断驱动的*read*方法的实现和我们已经看到的*write*方法的实现非常类似。来自设备的数据应该首先被读取到驱动程序的缓冲区，然后当缓冲区中已经积累了足够多数量的数据时，或者完整的读取请求可被满足时，或者某种类型的超时发生时，再将这些数据复制到用户空间。

快速参考

本章介绍了与中断管理相关的符号：

```
#include <linux/interrupt.h>
int request_irq(unsigned int irq, irqreturn_t (*handler)(), unsigned long
  flags, const char *dev_name, void *dev_id);
void free_irq(unsigned int irq, void *dev_id);
```

上面这些调用用来注册和注销中断处理例程。

```
#include <linux/irq.h.h>
int can_request_irq(unsigned int irq, unsigned long flags);
```

上述函数只在i386和x86_64体系架构上可用。当试图分配某个给定中断线的请求成功时，则返回非零值。

```
#include <asm/signal.h>
SA_INTERRUPT
SA_SHIRQ
SA_SAMPLE_RANDOM
```

*request_irq*函数的标志。SA_INTERRUPT要求安装一个快速的处理例程（相对于慢

速的)。SA_SHIRQ安装一个共享的处理例程，而第三个标志表明中断时间戳可用来产生系统熵。

```
/proc/interrupts
/proc/stat
```

这些文件系统节点用于汇报关于硬件中断和已安装处理例程的信息。

```
unsigned long probe_irq_on(void);
int probe_irq_off(unsigned long);
```

当驱动程序不得不探测设备，以确定该设备使用哪根中断信号线时，可以使用上述函数。在中断产生之后，*probe_irq_on* 的返回值必须传回给 *probe_irq_off*，而 *probe_irq_off* 的返回值就是检测到的中断号。

```
IRQ_NONE
IRQ_HANDLED
IRQ_RETVAL(int x)
```

中断处理例程的可能返回值，它们表示是否是一个真正来自设备的中断。

```
void disable_irq(int irq);
void disable_irq_nosync(int irq);
void enable_irq(int irq);
```

驱动程序可以启用和禁用中断报告。如果硬件试图在中断被禁用的时候产生中断，中断将永久丢失。使用共享处理例程的驱动程序不能使用这些函数。

```
void local_irq_save(unsigned long flags);
void local_irq_restore(unsigned long flags);
```

使用 *local_irq_save* 可禁用本地处理器上的中断，并记录先前的状态。flags可传递给 *local_irq_restore* 以恢复先前的中断状态。

```
void local_irq_disable(void);
void local_irq_enable(void);
```

用于无条件禁用和启用当前处理器中断的函数。



第十一章

内核的数据类型

在继续讨论更高级的主题之前，我们需要先讨论一下可移植性问题。现代版本的Linux内核的可移植性是非常好的，可以运行在许多不同的体系架构上。由于Linux的多平台特性，任何一个重要的驱动程序都应该都是可移植的。

但是与内核代码相关的核心问题是这些代码应该能够同时访问已知长度（例如，文件系统的数据结构或者设备板上的寄存器）的数据项，并充分利用不同处理器（32位和64位体系架构，或者也可能是16位的）的能力。

在把x86上的代码移植到新的体系架构上时，内核开发人员遇到的若干问题都和不正确的数据类型有关。坚持使用严格的数据类型，并且使用 *-Wall -Wstrict-prototypes* 选项编译可以防止大多数的代码缺陷。

内核使用的数据类型主要被分成三大类：类似`int`这样的标准C语言类型，类似`u32`这样的有确定大小的类型，以及像`pid_t`这样的用于特定内核对象的类型。我们将讨论应该在什么情况下使用这三种典型类型，以及如何使用。当从x86平台向其他平台移植驱动程序代码时，读者可能遇到其他一些典型的问题，这些问题将在本章的最后一节讨论。本章还将介绍新内核头文件提供的对链表的通用支持。

如果读者遵循我们提供的指导方针，读者的驱动程序甚至可能在那些未经测试的平台上编译和运行。

使用标准C语言类型

尽管大多数程序员习惯于自由使用像`int`和`long`这样的标准类型，但编写设备驱动程序时需要小心，以避免类型冲突和潜在的代码缺陷。

问题是，当我们需要“两个字节的填充符”或者“用四个字节字符串表示的某个东西”

时，我们不能使用标准类型，因为在不同的体系架构上，普通C语言的数据类型所占空间的大小并不相同。在O'Reilly ftp站点上的*misc-progs*目录下提供的样例文件包含了*datasize*程序，它可以显示各种C语言数据类型的大小。以下是PC上该程序的运行样例（其中最后四个类型将在下一节介绍）：

```
morgana% misc-progs/datasize
arch   Size:  char  short  int  long   ptr long-long  u8 u16 u32 u64
i686           1     2     4     4     4      8        1   2   4   8
```

这个程序也可以在64位平台上运行，其结果表明在64位系统上long整型和指针类型的大小和32位系统不同。下面的结果说明了该程序在不同平台上的运行结果：

```
arch   Size:  char  short  int  long   ptr long-long  u8 u16 u32 u64
i386           1     2     4     4     4      8        1   2   4   8
alpha          1     2     4     8     8      8        1   2   4   8
armv4l         1     2     4     4     4      8        1   2   4   8
ia64           1     2     4     8     8      8        1   2   4   8
m68k           1     2     4     4     4      8        1   2   4   8
mips           1     2     4     4     4      8        1   2   4   8
ppc            1     2     4     4     4      8        1   2   4   8
sparc          1     2     4     4     4      8        1   2   4   8
sparc64        1     2     4     4     4      8        1   2   4   8
x86_64         1     2     4     8     8      8        1   2   4   8
```

值得注意的是，SPARC 64架构运行的是32位的用户空间，因此在用户空间指针是32位宽的，不过它们在内核空间是64位的。这可以通过装载*kdatasize*模块（可从*misc-modules*目录下的样例文件中得到）来验证。该模块在装载时使用*printk*来报告大小信息并返回一个错误（所以不需要卸载这个模块）：

```
kernel: arch   Size:  char short int long  ptr long-long u8 u16 u32 u64
kernel: sparc64         1    2    4    8    8     8       1   2   4   8
```

尽管在混合使用不同数据类型时我们必须小心谨慎，但有时有理由这样做。这样的一种情况是内存地址，只要一涉及到内核，内存地址就变得很特殊。虽然从概念上讲地址是指针，但是通过使用无符号整数类型可以更好地实现内存管理；内核把物理内存看作是一个巨型数组，一个内存地址就是该数组的一个索引。此外，我们可以很方便地对指针取值；但在直接处理内存地址时，我们几乎从来不会以这种方式对它们取值。使用一个整数类型可以防止这种取值，因而可避免代码缺陷。所以，内核中的普通内存地址通常是unsigned long，这利用了如下事实：至少在当前Linux支持的所有平台上，指针和long整型的大小总是相同的。

C99标准定义了intptr_t和uintptr_t类型，它们是能够保存指针值的整型变量。这些类型在2.6的内核中几乎没有用到。

为数据项分配确定的空间大小

有时内核代码需要特定大小的数据项，多半是用来匹配预定义的二进制结构（注1），或者和用户空间进行通信，或者通过在结构体中插入“填白（padding）”字段（关于对齐的问题，请查阅“数据对齐”一节）来对齐数据。

当我们需要知道自己的数据大小时，内核提供了下列数据类型。所有这些类型都在头文件 *<asm/types.h>* 中声明，这个文件又被头文件 *<linux/types.h>* 包含：

```
u8;   /* 无符号字节(8位) */
u16;  /* 无符号字(16位) */
u32;  /* 无符号 32 位值 */
u64;  /* 无符号 64 位值 */
```

相应的有符号类型也存在，但是几乎没用。如果需要它们的话，只需将名字中的u用s替换就可以了。

如果一个用户空间程序需要使用这些类型，它可以在名字前加上两个下划线作为前缀：__u8和其他类型是独立于__KERNEL__定义的。例如，如果一个驱动程序需要通过*ioctl*系统调用与一个运行在用户空间的程序交换二进制结构的话，则应该在头文件中用__u32来声明结构中的32位的成员。

重要的是要记住这些类型是Linux特有的，使用它们将阻碍软件向其他Unix变种的移植。使用新编译器的系统将支持C99标准类型，例如uint8_t和uint32_t；如果考虑到可移植性，可以使用这些类型而不是Linux特有的变种。

我们可能也会注意到有时内核使用传统的类型，例如unsigned int，这通常用于其大小与体系架构无关的数据项。这种做法通常是为了保持向后兼容性。当在版本1.1.67中引入u32及其相关类型时，开发者没有办法将现存的数据结构改变为新的类型，因为当结构体字段和所赋予的值之间类型不匹配时，编译器将发出警告（注2）。Linus没有想到他自己编写的操作系统会用在多平台上，因此，旧的结构体有时定义得不是很严格。

接口特定的类型

内核中最常用的数据类型由它们自己的typedef声明，这样可以防止出现任何移植性问题。例如，一个进程的标识符（pid）通常使用pid_t类型，而不是int。使用pid_t

注1：当读取分区表，执行二进制文件，或者解开网络数据包时会发生这种情况。

注2：事实上，即使两个相同的对象具有不同的类型名称时，编译器信号类型也会矛盾，比如PC上的unsigned long和u32。

屏蔽了在实际的数据类型中任何可能的差异。“接口特定（interface-specific）”是指由某个库定义的一种数据类型，以便为某个特定的数据结构提供接口。

注意，近来已经很少定义新的接口特定的类型。许多内核开发人员已经不再喜欢使用typedef语句，他们更愿意看到直接用在代码中的真实的类型信息，而不是隐藏在用户定义的类型之后。不过，许多较老的接口特定类型还是保留在内核中，它们不会很快就消失。

即使没有定义接口特定的类型，也应该始终使用和内核其余部分一致的、适当的数据类型。例如，jiffies计数总是属于unsigned long类型，而不管它的实际大小如何，因此，在使用jiffies的时候应该始终使用unsigned long类型。本节中我们将主要讨论“_t”类型的用法。

完整的_t类型在*<linux/types.h>*中定义，但很少使用这个清单。当需要某个特定类型时，可在所需调用的函数原型或者所使用的数据结构中找到这个类型。

只要驱动程序使用了需要这种“定制”类型的函数，但又不遵守约定的时候，编译器就会产生警告；如果使用*-Wall*编译选项并且细心地消除所有的警告，就可以确信代码是可移植的了。

_t数据项的主要问题是在我们需要打印它们的时候，不太容易选择正确的*printk*或者*printf*的输出格式，并且在一种体系架构上排除的警告，在另一种体系架构上可能还会出现。例如，当size_t在一些平台上是unsigned long，而在另一些平台上是unsigned int类型时，我们应该如何打印它呢？

当我们需要打印一些接口特定的数据类型时，最行之有效的方法，就是将其强制转换成可能的最大类型（通常是long或者unsigned long），然后用相应的格式打印。这种做法不会产生错误或者警告，因为格式和类型相匹配，而且也不会丢失数据位，因为强制类型转换要么是一个空操作，要么是将该数据项向更宽的数据类型扩展。

实际上，通常并不需要打印我们讨论的这些数据项，因此，只有在调试信息中才会出现这些问题。除了将接口特定的数据类型作为参数传递给库函数或者内核函数之外，大多数时候代码只需对它们进行储存和比较操作。

尽管_t类型在大多数情况下是正确的解决方案，但有时正确的类型并不存在。这发生在一些还没有被整理的旧接口上。

在内核头文件中我们已经发现一处疑点，I/O函数的数据类型不是很严格（请参阅第九章的“平台相关性”一节）。这种不严格的类型定义主要是由于历史原因造成的，但是

却可能在编写代码时引起问题。例如，在把参数交换给像*outb*这样的函数时经常遇到麻烦；如果有一种port_t类型，编译器就会发现这一类错误。

其他有关移植性的问题

在编写一个能在不同的Linux平台间移植的驱动程序时，除了数据类型定义的问题之外，还必须注意其他一些软件上的问题。

一个通用的原则是要避免使用显式的常量值。通常，代码通过使用预处理的宏使之参数化。这一节列出了最重要的移植性问题。在遇到其他已经被参数化的值时，可以在头文件和随正式内核版本一起发布的设备驱动程序中找到一些线索。

时间间隔

在处理时间间隔时，不要假定每秒一定有100个jiffies。尽管对于当前的i386架构这是正确的，但并不是每一种Linux平台都以这个速度运行。即使在x86上这种假设也可能是错误的，因为HZ值可能已被改变（有这种情况），更何况没有人知道未来的内核将发生什么变化。使用jiffies计算时间间隔的时候，应该用HZ（每秒定时器中断的次数）来衡量。例如，为了检测半秒的超时，可以将消逝的时间与HZ/2作比较。更常见的，与msec毫秒对应的jiffies数目总是msec*HZ/1000。

页大小

使用内存时，要记住内存页的大小为PAGE_SIZE字节，而不是4 KB。假定页大小就是4 KB而且硬编码这个数值，是PC程序员常犯的错误。相反，在已支持的平台上，页大小范围是从4 KB到64 KB，有时候它们在相同平台的不同实现上也是不一致的。这一问题涉及到的宏是PAGE_SIZE和PAGE_SHIFT。后者是为得到一个地址所在页的页号，需要对该地址右移的位数。对于4 KB和更大的页，这个数值通常是12或者更大。这些宏在头文件*<asm/page.h>*中定义；如果用户空间程序需要这些信息，则可以使用*getpagesize*库函数来获得。

让我们看看一种重要的情形。如果一个驱动程序需要16 KB空间来储存临时数据，我们不应该指定传递给*get_free_pages*的参数为2的幂，而需要一个可移植的方案。幸运的是，内核开发人员已经编写了一个名为*get_order*的解决方案：

```
#include <asm/page.h>
int order = get_order(16*1024);
buf = get_free_pages(GFP_KERNEL, order);
```

记住传递给*get_order*的参数必须是2的幂。

字节序

小心不要做字节序的假设。尽管PC是按照先是低字节（little-endian，小头）的方式存储多字节数值的，但某些高端平台是以另一种方式（big-endian，大头）工作的。只要可能，就应该将代码编写成不依赖于所操作数据的字节序的方式。可是，有时驱动程序需要从单字节建立整型数或者相反，或者它必须和要求特定字节序的设备通信。

头文件<*asm/byteorder.h*>定义了__BIG_ENDIAN或者__LITTLE_ENDIAN，取决于处理器的字节序。在处理字节序问题时，我们可能要编写一组#ifdef __LITTLE_ENDIAN条件语句，但是有一个更好的方法。Linux内核定义了一组宏，它可以在处理器字节序和特殊字节序之间进行转换。例如：

```
u32 cpu_to_le32 (u32);
u32 le32_to_cpu (u32);
```

这两个宏将一个CPU使用的值转换成一个无符号的32位小头数值，或者相反。不管CPU是大头还是小头它们都可以正常工作，也不管CPU是否是一个32位处理器。如果没有转换工作需要做，它们就返回未经修改的参数。使用这些宏可以使编写可移植代码的工作变得更加容易，而无需使用很多条件编译。

类似的例程有十几个之多，我们可以在头文件<*linux/byteorder/big_endian.h*>和<*linux/byteorder/little_endian.h*>中看到完整的列表。稍后就会看到，这种模式很容易遵循。*be64_to_cpu*将一个无符号的64位大头的数值转换成CPU的内部表示形式。相应地，*le16_to_cpus*处理一个有符号的16位小头的数值。当涉及到指针时，也可以使用类似*cpu_to_le32p*这样的函数，它们以指向数值的指针而不是数值本身为参数。其他函数可参阅头文件。

数据对齐

编写可移植代码的最后一个值得关注的问题是如何访问未对齐的数据，例如，怎样读取一个存储在非四字节倍数的地址中的四字节值。i386的用户常常访问未对齐的数据项，但不是所有的体系架构都允许这样做的。大部分现代体系架构在每次程序试图传输未对齐的数据时都会产生一个异常，这时，数据传输会被异常处理程序处理，因此会带来大量性能损失。如果需要访问未对齐的数据，则应该使用下面的宏：

```
#include <asm/unaligned.h>
get_unaligned(ptr);
put_unaligned(val, ptr);
```

这些宏是与类型无关的，对各种数据项，不管它是1字节、2字节、4字节还是8字节，这些宏都有效。所有版本的内核都定义了这些宏。

另一个关于对齐的问题是数据结构的跨平台可移植性。同样的数据结构（在C语言源文件中定义的）在不同的平台上可能会被编译成不同的布局。编译器根据平台的习惯来对齐数据结构的字段，而不同平台的习惯是不同的。

为了编写可以在不同平台之间可移植的数据项的数据结构，除了规定特定的字节序以外，还应该始终强制数据项的自然对齐。自然对齐（natural alignment）是指在数据项大小的整数倍（例如，8字节数据项存入8的整数倍的地址）的地址处存储数据项。强制自然对齐可以防止编译器移动数据结构的字段，你应该使用填充符（filler）字段来避免在数据结构中留下空洞。

为说明编译器是怎样强制对齐的，示例源代码的*misc-progs*目录中有个*dataalign*程序，对应的内核模块是*misc-modules*目录中的*kdataalign*。下面是*dataalign*程序在若干平台上的输出，以及*kdataalign*模块在SPARC64上的输出：

```
arch  Align:  char  short  int  long   ptr long-long  u8 u16 u32 u64
i386            1      2    4     4     4         4   1   2   4   4
i686            1      2    4     4     4         4   1   2   4   4
alpha           1      2    4     8     8         8   1   2   4   8
armv4l          1      2    4     4     4         4   1   2   4   4
ia64            1      2    4     8     8         8   1   2   4   8
mips            1      2    4     4     4         8   1   2   4   8
ppc             1      2    4     4     4         8   1   2   4   8
sparc           1      2    4     4     4         8   1   2   4   8
sparc64         1      2    4     4     4         8   1   2   4   8
x86_64          1      2    4     8     8         8   1   2   4   8

kernel: arch  Align: char short int long  ptr long-long u8 u16 u32 u64
kernel: sparc64        1    2    4    8    8         8       1   2   4   8
```

值得注意的是，不是所有平台都在64位边界对齐64位数值，所以需要填充符字段来强制对齐并确保可移植性。

最后，要注意编译器本身也许会悄悄地往结构体中插入填充数据，来确保每个字段的对齐可以在目标处理器上取得好的性能。如果正在定义一个和设备所要求的结构体相匹配的结构体，这种自动填充会破坏你的意图。解决办法是告诉编译器该结构体必须是“填满的”，不能添加填充符。例如，内核头文件*<linux/edd.h>*定义了几个用于和x86 BIOS交互的数据结构，包括如下定义：

```
struct {
        u16 id;
        u64 lun;
        u16 reserved1;
        u32 reserved2;
} __attribute__ ((packed)) scsi;
```

如果没有__attribute__ ((packed))，lun字段前面会被插入两个填充字节，如果在64位平台上编译该结构体的话就是6个字节。

指针和错误值

许多内部的内核函数返回一个指针值给调用者，而这些函数中很多可能会失败。在大部分情况下，失败是通过返回一个NULL指针值来表示的。这种技巧有作用，但是它不能传递问题的确切性质。某些接口确实需要返回一个实际的错误编码，以使调用者可以根据实际出错的情况做出正确的决策。

许多内核接口通过把错误值编码到一个指针值中来返回错误信息。这种函数必须小心使用，因为它们的返回值不能简单地和NULL比较。为了帮助创建和使用这种类型的接口，*<linux/err.h>*中提供了一小组函数。

返回指针类型的函数可以通过如下函数返回一个错误值：

```
void *ERR_PTR(long error);
```

这里error是通常的负的错误编码。调用者可以使用*IS_ERR*来检查所返回的指针是否是一个错误编码：

```
long IS_ERR(const void *ptr);
```

如果需要实际的错误编码，可以通过如下函数把它提取出来：

```
long PTR_ERR(const void *ptr);
```

应该只有在*IS_ERR*对某值返回真值时才对该值使用*PTR_ERR*，因为任何其他值都是有效的指针。

链表

就像很多其他程序一样，操作系统内核经常需要维护数据结构的列表。有时，Linux内核中同时存在多个链表的实现代码。为了减少重复代码的数量，内核开发者已经建立了一套标准的循环、双向链表的实现。如果你需要操作链表，那么建议你使用这一内核设施。

当使用这些链表接口时，应该始终牢记这些链表函数不进行任何锁定。如果你的驱动程序有可能试图对同一个链表执行并发操作的话，则有责任实现一个锁方案。否则，崩溃的链表结构体、数据丢失、内核混乱等问题是很难诊断的。

为了使用这个链表机制，驱动程序必须包含头文件<*linux/list.h*>。该文件定义了一个简单的`list_head`类型的结构体。

```
struct list_head {
    struct list_head *next, *prev;
};
```

用于实际代码的链表几乎总是由某种结构类型构成，每个结构描述链表中的一项。为了在代码中使用Linux链表设施，只需要在构成链表的结构里面嵌入一个`list_head`。如果驱动程序维护一个链表，则可声明如下：

```
struct todo_struct {
    struct list_head list;
    int priority; /* 驱动程序特定的 */
    /* ... 增加其他驱动程序特定的字段 */
};
```

链表头通常是一个独立的`list_head`结构。图11-1显示了简单的`struct list_head`是如何用来维护一个数据结构链表的。

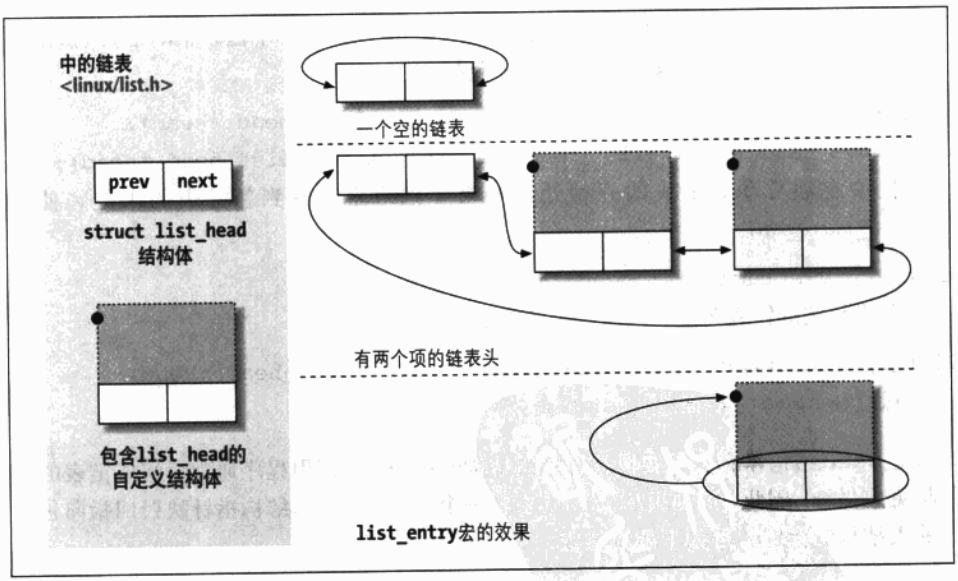


图 11-1: list_head 数据结构

在使用之前，必须用INIT_LIST_HEAD宏来初始化链表头。可如下声明并初始化一个实际的链表头：

```
struct list_head todo_list;

INIT_LIST_HEAD(&todo_list);
```

另外，可在编译时像下面这样初始化链表：

```
LIST_HEAD(todo_list);
```

头文件 <*linux/list.h*> 中声明了下列操作链表的函数：

`list_add(struct list_head *new, struct list_head *head);`

在链表头后面添加新项 —— 通常是在链表的头部。这样，它可以被用来建立栈。但需要注意的是，head并不一定非得是链表名义上的头；如果传递了一个恰巧位于链表中间某处的list_head结构体，新项会紧跟在它的后面。因为Linux链表是循环式的，链表头通常与其他的项没有本质上的区别。

`list_add_tail(struct list_head *new, struct list_head *head);`

在给定链表头的前面添加一个新的项，即在链表的末尾处添加。因此，可使用 *list_add_tail* 来建立先进先出（FIFO）队列。

`list_del(struct list_head *entry);`
`list_del_init(struct list_head *entry);`

删除链表中的给定项。如果该项还可能被重新插入到另一个链表中的话，应该使用 *list_del_init*，它会重新初始化链表的指针。

`list_move(struct list_head *entry, struct list_head *head);`
`list_move_tail(struct list_head *entry, struct list_head *head);`

把给定项移动到链表的开始处。如果要把给定项放到新链表的末尾，使用 *list_move_tail*。

`list_empty(struct list_head *head);`

如果给定的链表为空，返回一个非零值。

`list_splice(struct list_head *list, struct list_head *head);`

通过在head之后插入list来合并两个链表。

list_head结构体有利于实现具有类似结构的链表，但调用程序通常对组成链表的大结构更感兴趣。因此，可利用 *list_entry* 宏将一个list_head结构指针映射回指向包含它的大结构的指针。可如下调用该宏：

```
list_entry(struct list_head *ptr, type_of_struct, field_name);
```

其中，ptr是指向正被使用的struct list_head的指针，type_of_struct是包含ptr的结构类型，field_name是结构中链表字段的名字。在之前的todo_struct结构中，链表字段只是简单地被称为list。这样，利用类似下面的代码行，我们可以将一个链表项转换成包含它的结构：

```
struct todo_struct *todo_ptr =
    list_entry(listptr, struct todo_struct, list);
```

需要稍微习惯一下宏*list_entry*，但还不是很难使用。

遍历链表很容易：只需跟随prev和next指针。作为例子，假设我们想让todo_struct链表中的项按照优先级降序排列，则增加新项的函数如下所示：

```
void todo_add_entry(struct todo_struct *new)
{
    struct list_head *ptr;
    struct todo_struct *entry;

    for (ptr = todo_list.next; ptr != &todo_list; ptr = ptr->next) {
        entry = list_entry(ptr, struct todo_struct, list);
        if (entry->priority < new->priority) {
            list_add_tail(&new->list, ptr);
            return;
        }
    }
    list_add_tail(&new->list, &todo_struct)
}
```

然而，作为一个惯例，最好使用一组预定义的宏来创建可以遍历链表的循环。例如，前一个循环可以如下编码：

```
void todo_add_entry(struct todo_struct *new)
{
    struct list_head *ptr;
    struct todo_struct *entry;

    list_for_each(ptr, &todo_list) {
        entry = list_entry(ptr, struct todo_struct, list);
        if (entry->priority < new->priority) {
            list_add_tail(&new->list, ptr);
            return;
        }
    }
    list_add_tail(&new->list, &todo_struct)
}
```

使用所提供的宏有助于避免简单的编程错误，而且这些宏的开发人员还对它们的性能进行了一定的优化。下面是一些变体：

`list_for_each(struct list_head *cursor, struct list_head *list)`

该宏创建一个for循环，每当游标指向链表中的下一项时执行一次。在遍历链表时要注意对它的修改。

`list_for_each_prev(struct list_head *cursor, struct list_head *list)`

该版本向后遍历链表。

`list_for_each_safe(struct list_head *cursor, struct list_head *next, struct list_head *list)`

如果循环可能会删除链表中的项，就应该使用该版本。它只是简单地在循环的开始处把链表中的下一项存储在next中，这样如果cursor所指的项被删除也不会造成混乱。

`list_for_each_entry(type *cursor, struct list_head *list, member)`
`list_for_each_entry_safe(type *cursor, type *next, struct list_head *list, member)`

这些宏使处理一个包含给定类型结构体的链表时更加容易。这里，cursor是指向包含结构体类型的指针，member是包含结构体内list_head结构体的名字。使用这些宏就不需要在循环内调用list_entry了。

如果你查看*<linux/list.h>*，就会发现一些额外的声明。hlist类型是一个使用分离的、单指针链表头类型的双向链表；它经常用于创建散列表和类似的数据结构。还有用于遍历这两种类型链表的宏，它们一般用于“读－复制－更新”机制（在第五章的“读取－复制－更新”一节中介绍）。这些基本例程不太可能用于设备驱动程序，但如果读者想深入探究的话可以查阅该头文件。

快速参考

本章介绍了如下符号：

`#include <linux/types.h> typedef u8;`
`typedef u16;`
`typedef u32;`
`typedef u64;`

确保是8、16、32和64位的无符号整数值类型。对应的有符号类型同样存在。在用户空间，读者可以使用__u8和__u16等类型。

`#include <asm/page.h>`
`PAGE_SIZE`
`PAGE_SHIFT`

定义了当前体系架构的每页字节数和页偏移位数（4 KB页为12、8 KB页为13）的符号。

`#include <asm/byteorder.h>`
`__LITTLE_ENDIAN`
`__BIG_ENDIAN`

这两个符号只有一个被定义，取决于体系架构。

```
#include <asm/byteorder.h>
u32 __cpu_to_le32 (u32);
u32 __le32_to_cpu (u32);
```

在已知字节序和处理器字节序之间进行转换的函数。有超过 60 个这样的函数；关于它们的完整列表和如何定义，请查阅 *include/linux/byteorder/* 下的各种文件。

```
#include <asm/unaligned.h>
get_unaligned(ptr);
put_unaligned(val, ptr);
```

某些体系架构需要使用这些宏来保护未对齐的数据。对于允许访问未对齐数据的体系架构，这些宏扩展为普通的指针取值。

```
#include <linux/err.h>
void *ERR_PTR(long error);
long PTR_ERR(const void *ptr);
long IS_ERR(const void *ptr);
```

这些函数允许从返回指针值的函数中获得错误编码。

```
#include <linux/list.h>
list_add(struct list_head *new, struct list_head *head);
list_add_tail(struct list_head *new, struct list_head *head);
list_del(struct list_head *entry);
list_del_init(struct list_head *entry);
list_empty(struct list_head *head);
list_entry(entry, type, member);
list_move(struct list_head *entry, struct list_head *head);
list_move_tail(struct list_head *entry, struct list_head *head);
list_splice(struct list_head *list, struct list_head *head);
```

操作循环、双向链表的函数。

```
list_for_each(struct list_head *cursor, struct list_head *list)
list_for_each_prev(struct list_head *cursor, struct list_head *list)
list_for_each_safe(struct list_head *cursor, struct list_head *next, struct
  list_head *list)
list_for_each_entry(type *cursor, struct list_head *list, member)
list_for_each_entry_safe(type *cursor, type *next struct list_head *list,
  member)
```

遍历链表的便利宏。

第十二章

PCI 驱动程序



第九章介绍了最底层的硬件控制，而本章将给出一个高层总线架构的综述。总线由电气接口和编程接口构成。本章将重点讨论编程接口。

本章涉及到许多总线架构。不过，我们的讨论重点是用于访问 Peripheral Component Interconnect（PCI，外围设备互联）外设的内核函数，因为PCI总线是当今普遍使用在桌面以及更大型计算机上的外设总线，而且该总线是内核中得到最好支持的总线。虽然ISA总线基本上是一种“裸金属”类型的总线，但在电子爱好者中仍然很常用，本章稍后将进行介绍。不过，除了我们在第九章和第十章中涉及的内容以外，已没有太多需要介绍的了。

PCI 接口

尽管许多计算机用户将PCI看成是一种布置电子线路的方式，但实际上它是一组完整的规范，定义了计算机的各个不同部分之间应该如何交互。

PCI 规范涵盖了与计算机接口相关的大部分问题。我们不打算在这里讲述所有的内容；本节主要介绍 PCI 驱动程序如何寻找其硬件和获得对它的访问。第二章的“模块参数”一节以及第十章的“自动检测 IRQ 编号”一节所讨论的探测技术，也可用于 PCI 设备，但是 PCI 规范提供了一种更好的探测方法。

PCI 架构被设计为 ISA 标准的替代品，它有三个主要目标：获得在计算机和外设之间传输数据时更好的性能；尽可能的平台无关；简化往系统中添加和删除外设的工作。

通过使用比ISA更高的时钟频率，PCI总线获得了更好的性能；它的时钟频率一般是25或者33 MHz（实际的频率是系统时钟的系数），最新的实现达到了66 MHz甚至133 MHz。此外，它配备了32位的数据总线，而且规范已经包括了64位的扩展。平台无关

性通常也是计算机总线的一个设计目标，对PCI来说平台无关性尤其重要，因为PC世界一直以来都是由一些处理器特有的接口标准所主宰。目前，PCI广泛应用于IA-32、Alpha、PowerPC、SPARC64和IA-64等系统中。

和驱动程序编写者息息相关的问题是接口板的自动检测。PCI设备是无跳线设备（不像大部分的老式外设），可在引导阶段自动配置。这样，设备驱动程序必须能够访问设备中的配置信息以便完成初始化。对于PCI设备来说，这些工作无需探测就能完成。

PCI寻址

每个PCI外设由一个总线编号、一个设备编号及一个功能编号来标识。PCI规范允许单个系统拥有高达256个总线，但是因为256个总线对于许多大型系统而言是不够的，因此，Linux目前支持PCI域。每个PCI域可以拥有最多256个总线。每个总线上可支持32个设备，而每个设备都可以是多功能板（例如音频设备外加CD-ROM驱动器），最多可有八种功能。所以，每种功能都可以在硬件级由一个16位的地址（或键）来标识。不过，为Linux编写的设备驱动程序无需处理这些二进制的地址，因为它们使用一种特殊的数据结构（名为`pci_dev`）来访问设备。

当前的工作站一般配置有至少两个PCI总线。在单个系统中插入多个总线，可通过桥（bridge）来完成，它是用来连接两个总线的特殊PCI外设。PCI系统的整体布局组织为树型，其中每个总线连接到上一级总线，直到树根的0号总线。CardBus PC卡系统也是通过桥连接到PCI系统的。典型的PCI系统可见图12-1，其中标记出了各种不同的桥。

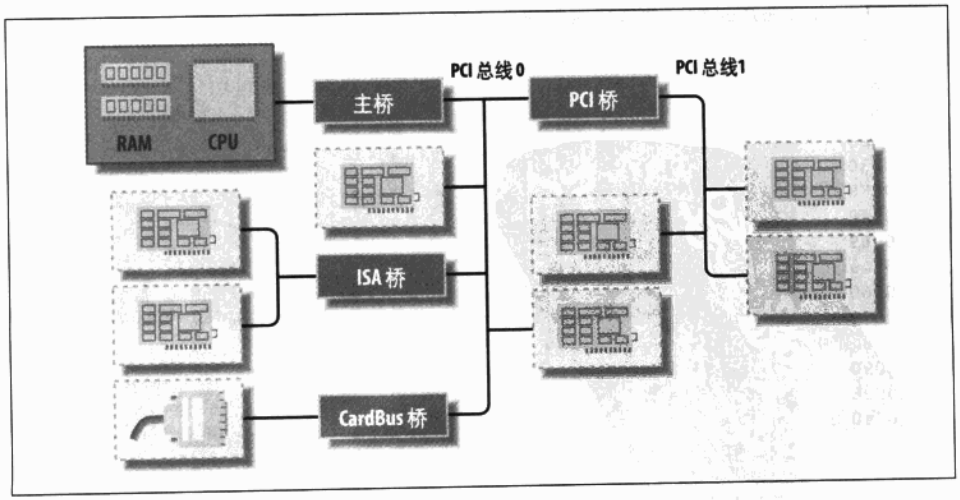


图 12-1：典型 PCI 系统的布局

尽管和PCI外设关联的16位硬件地址通常隐藏在struct pci_dev对象中，但有时仍然可见，尤其是这些设备正在被使用时。*lspci*（*pciutils*包的一部分，包含在大多数发行版中）的输出以及*/proc/pci*和*/proc/bus/pci*中信息的布局就是这种情况。PCI设备在sysfs中的表示同样展现了这种寻址方案，此外还有PCI域的信息（注1）。在显示硬件地址时，有时显示为两个值（一个8位的总线编号和一个8位的设备及功能编号），有时显示为三个值（总线、设备和功能），有时显示为四个值（域、总线、设备和功能）；所有的值通常都以16进制显示。

例如，*/proc/bus/pci/devices*使用单个16位字段（便于解析及排序），而*/proc/bus/busnumber*将地址划分成了三个字段。下面说明了这些地址如何出现，只列出了输出行的开始部分：

```
$ lspci | cut -d: -f1-3
0000:00:00.0 Host bridge
0000:00:00.1 RAM memory
0000:00:00.2 RAM memory
0000:00:02.0 USB Controller
0000:00:04.0 Multimedia audio controller
0000:00:06.0 Bridge
0000:00:07.0 ISA bridge
0000:00:09.0 USB Controller
0000:00:09.1 USB Controller
0000:00:09.2 USB Controller
0000:00:0c.0 CardBus bridge
0000:00:0f.0 IDE interface
0000:00:10.0 Ethernet controller
0000:00:12.0 Network controller
0000:00:13.0 FireWire (IEEE 1394)
0000:00:14.0 VGA compatible controller
$ cat /proc/bus/pci/devices | cut -f1
0000
0001
0002
0010
0020
0030
0038
0048
0049
004a
0060
0078
0080
0090
0098
00a0
```

注1：某些体系架构也会在*/proc/pci*和*/proc/bus/pci*文件中显示PCI域信息。

```
$ tree /sys/bus/pci/devices/
/sys/bus/pci/devices/
|-- 0000:00:00.0 -> ../../../devices/pci0000:00/0000:00:00.0
|-- 0000:00:00.1 -> ../../../devices/pci0000:00/0000:00:00.1
|-- 0000:00:00.2 -> ../../../devices/pci0000:00/0000:00:00.2
|-- 0000:00:02.0 -> ../../../devices/pci0000:00/0000:00:02.0
|-- 0000:00:04.0 -> ../../../devices/pci0000:00/0000:00:04.0
|-- 0000:00:06.0 -> ../../../devices/pci0000:00/0000:00:06.0
|-- 0000:00:07.0 -> ../../../devices/pci0000:00/0000:00:07.0
|-- 0000:00:09.0 -> ../../../devices/pci0000:00/0000:00:09.0
|-- 0000:00:09.1 -> ../../../devices/pci0000:00/0000:00:09.1
|-- 0000:00:09.2 -> ../../../devices/pci0000:00/0000:00:09.2
|-- 0000:00:0c.0 -> ../../../devices/pci0000:00/0000:00:0c.0
|-- 0000:00:0f.0 -> ../../../devices/pci0000:00/0000:00:0f.0
|-- 0000:00:10.0 -> ../../../devices/pci0000:00/0000:00:10.0
|-- 0000:00:12.0 -> ../../../devices/pci0000:00/0000:00:12.0
|-- 0000:00:13.0 -> ../../../devices/pci0000:00/0000:00:13.0
`-- 0000:00:14.0 -> ../../../devices/pci0000:00/0000:00:14.0
```

这三个设备清单以相同的顺序排列，因为*lspci*使用*/proc*文件作为其信息来源。以VGA视频控制器为例，当划分为域（16位）、总线（8位）、设备（5位）和功能（3位）时，`0x00a0`表示`0000:00:14.0`。

每个外设板的硬件电路对如下三种地址空间的查询进行应答：内存位置、I/O端口和配置寄存器。前两种地址空间由同一PCI总线上的所有设备共享（也就是说，在访问内存位置时，该PCI总线上的所有设备将在同一时间看到总线周期）。另一方面，配置空间利用了地理寻址（geographical addressing）。配置查询每次只对一个槽寻址，因此它们根本不会发生任何冲突。

对驱动程序而言，内存和I/O区域是以惯常的方式，即通过*inb*和*readb*等等进行访问的。另一方面，配置事务是通过调用特定的内核函数访问配置寄存器来执行的。关于中断，每个PCI槽有四个中断引脚，每个设备功能可使用其中的一个，而不用考虑这些引脚如何连接到CPU。这种路由是计算机平台的职责，实现在PCI总线之外。因为PCI规范要求中断线是可共享的，因此，即使是IRQ线有限的处理器（例如x86）仍然可以容纳许多PCI接口板（每个有四个中断引脚）。

PCI总线中的I/O空间使用32位地址总线（因此可有4 GB个端口），而内存空间可通过32位或64位地址来访问。64位地址在较新的平台上可用。通常假定地址对设备是唯一的，但是软件可能会错误地将两个设备配置成相同的地址，导致无法访问这两个设备。但是，如果驱动程序不去访问那些不应该访问的寄存器，就不会发生这样的问题。幸好，接口板提供的每个内存和I/O地址区域，都可以通过配置事务的方式进行重新映射。就是说，固件在系统引导时初始化PCI硬件，把每个区域映射到不同的地址以避免

冲突（注2）。这些区域所映射到的地址可从配置空间中读取，因此，Linux驱动程序不需要探测就能访问其设备。在读取配置寄存器之后，驱动程序就可以安全访问其硬件。

PCI配置空间中每个设备功能由256个字节组成（除了PCI快速设备以外，它的每个功能有4 KB的配置空间），配置寄存器的布局是标准化的。配置空间的4个字节含有一个独一无二的功能ID，因此，驱动程序可通过查询外设的特定ID来识别其设备（注3）。概言之，每个设备板是通过地理寻址来获取其配置寄存器的；这些寄存器中的信息随后可以被用来执行普通的I/O寻址，而不再需要额外的地理寻址。

到此应该清楚的是，PCI接口标准在ISA之上的主要创新在于配置地址空间。因此，除了通常的驱动程序代码之外，PCI驱动程序还需要访问配置空间的能力，以便免去冒险探测的工作。

在本章其余内容中，我们将使用“设备”一词来表示一种设备功能，因为多功能板上的每个功能都可以担当一个独立实体的角色。我们谈到设备时，表示的是一组“域编号、总线编号、设备编号和功能编号”。

引导阶段

为了解PCI的工作原理，我们需要从系统引导开始讲起，因为这是配置设备的阶段。

当PCI设备上电时，硬件保持未激活状态。换句话说，该设备只会对配置事务做出响应。上电时，设备上不会有内存和I/O端口映射到计算机的地址空间；其他设备相关功能，例如中断报告，也被禁止。

幸运的是，每个PCI主板均配备有能够处理PCI的固件，称为BIOS、NVRAM或PROM，这取决于具体的平台。固件通过读写PCI控制器中的寄存器，提供了对设备配置地址空间的访问。

系统引导时，固件（或者Linux内核，如果这样配置的话）在每个PCI外设上执行配置事务，以便为它提供的每个地址区域分配一个安全的位置。当驱动程序访问设备的时候，它的内存和I/O区域已经被映射到了处理器的地址空间。驱动程序可以修改这个默认配置，不过从来不需要这样做。

注2：实际上，配置并不限于系统引导阶段，比如热插拔设备在引导阶段并不存在，而是在后来才会出现。这里的要点是，设备驱动程序不能修改I/O和内存区域的地址。

注3：我们可从设备自己的硬件手册中找到ID。文件 *pci.ids* 中包含有一个清单，该文件是 *pciutils* 包和内核源代码的一部分。该文件并不完整，而只是列出了最著名的制造商及设备。该文件的内核版本将在未来的版本中删除。

我们曾经讲过，用户可以通过读取*/proc/bus/pci/devices*和*/proc/bus/pci/*/**来查看PCI设备清单和设备的配置寄存器。前者是个包含有十六进制的设备信息的文本文件，而后者是若干二进制文件，报告了每个设备的配置寄存器快照，每个文件对应一个设备。sysfs树中的个别PCI设备目录可以在*/sys/bus/pci/devices*中找到。一个PCI设备目录包含许多不同的文件：

```
$ tree /sys/bus/pci/devices/0000:00:10.0
/sys/bus/pci/devices/0000:00:10.0
|-- class
|-- config
|-- detach_state
|-- device
|-- irq
|-- power
|   `-- state
|-- resource
|-- subsystem_device
|-- subsystem_vendor
`-- vendor
```

*config*文件是一个二进制文件，使原始PCI配置信息可以从设备读取（就像*/proc/bus/pci/*/**所提供的）。*vendor*、*device*、*subsystem_device*、*subsystem_vendor*和*class*都表示该PCI设备的特定值（所有的PCI设备都提供这个信息）。*irq*文件显示分配给该PCI设备的当前IRQ，*resource*文件显示该设备所分配的当前内存资源。

配置寄存器和初始化

在本节中我们将查看PCI设备包含的配置寄存器。所有的PCI设备都有至少256字节的地址空间。前64字节是标准化的，而其余的是设备相关的。图12-2显示了设备无关的配置空间的布局。

如图12-2所示，某些PCI配置寄存器是必需的，而某些是可选的。每个PCI设备必须在必需的寄存器中包含有效值，而可选寄存器中的内容依赖于外设的实际功能。可选字段通常无用，除非必需字段的内容表明它们是有效的。这样，必需的字段声明了板子的功能，包括其他字段是否有用。

值得注意的是，PCI寄存器始终是小头的。尽管标准被设计为体系结构无关的，但PCI设计者仍然有点偏好PC环境。驱动程序编写者在访问多字节的配置寄存器时，要十分注意字节序，因为能够在PC上工作的代码到其他平台上可能就无法工作。Linux开发人员已经注意到了字节序问题（见下一节“访问配置空间”），但是这个问题必须被牢记心中。如果需要把数据从系统固有字节序转换成PCI字节序，或者相反，则可以借助定义在*<asm/byteorder.h>*中的函数，这些函数在第十一章中介绍过了，注意PCI字节序是小头的。

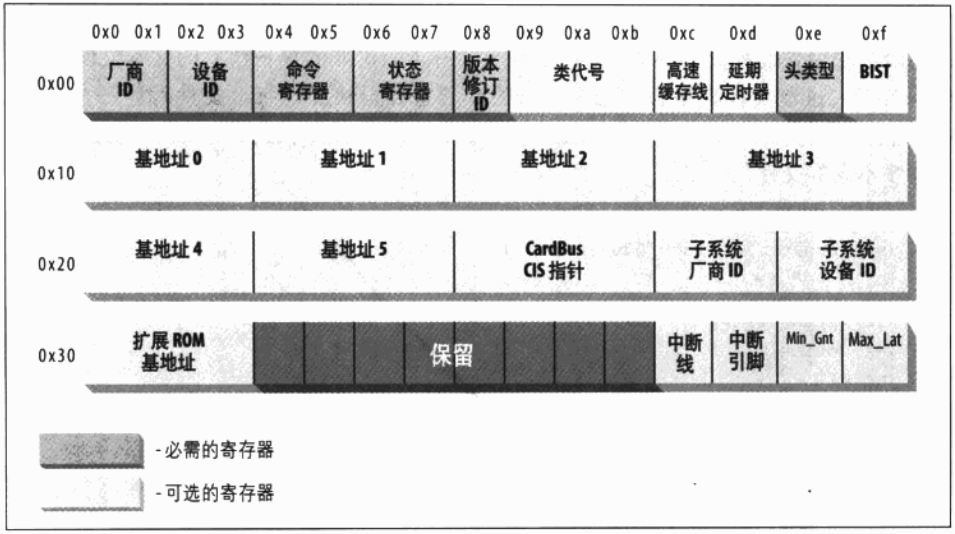


图 12-2: 标准化的 PCI 配置寄存器

对这些配置项的描述已经超过了本书讨论的范围。通常，随设备一同发布的技术文档会详细描述已支持的寄存器。我们所关心的是，驱动程序如何查询设备，以及如何访问设备的配置空间。

用三个或五个PCI寄存器可标识一个设备：vendorID、deviceID和class是常用的三个寄存器。每个PCI制造商会将正确的值赋予这三个只读寄存器，驱动程序可利用它们查询设备。此外，有时厂商利用subsystem vendorID和subsystem deviceID两个字段来进一步区分相似的设备。

下面是这些寄存器的详细介绍。

vendorID

这是一个16位的寄存器，用于标识硬件制造商。例如，每个Intel设备被标识为同一个厂商编号，即0x8086。PCI Special Interest Group维护有一个全球的厂商编号注册表，制造商必须申请一个唯一编号并赋于它们的寄存器。

deviceID

这是另外一个16位寄存器，由制造商选择；无需对设备ID进行官方注册。该ID通常和厂商ID配对生成一个唯一的32位硬件设备标识符。我们使用签名(signature)一词来表示一对厂商和设备ID。设备驱动程序通常依靠于该签名来识别其设备；可以从硬件手册中找到目标设备的签名值。

class

每个外部设备属于某个类(class)。class寄存器是一个16位的值，其中高8位标识了“基类(base class)”，或者组。例如，“ethernet(以太网)”和“token ring(令牌环)”是同属“network(网络)”组的两个类，而“serial(串行)”和“parallel(并行)”类同属“communication(通信)”组。某些驱动程序可支持多个相似的设备，每个具有不同的签名，但都属于同一个类；这些驱动程序可依靠class寄存器来识别它们的外设，如后所述。

subsystem vendorID

subsystem deviceID

这两个字段可用来进一步识别设备。如果设备中的芯片是一个连接到本地板载(onboard)总线上的通用接口芯片，则可能会用于完全不同的多种用途，这时，驱动程序必须识别它所关心的实际设备。子系统标识符就用于此目的。

PCI驱动程序可以使用这些不同的标识符来告诉内核它支持什么样的设备。struct pci_device_id结构体用于定义该驱动程序支持的不同类型的PCI设备列表。该结构体包含下列字段：

__u32 vendor;

__u32 device;

它们指定了设备的PCI厂商和设备ID。如果驱动程序可以处理任何厂商或者设备ID，这些字段应该使用值PCI_ANY_ID。

__u32 subvendor;

__u32 subdevice;

它们指定设备的PCI子系统厂商和子系统设备ID。如果驱动程序可以处理任何类型的子系统ID，这些字段应该使用值PCI_ANY_ID。

__u32 class;

__u32 class_mask;

这两个值使驱动程序可以指定它支持一种PCI类(class)设备。PCI规范中描述了不同类的PCI设备(例如VGA控制器)。如果驱动程序可以处理任何类型的子系统ID，这些字段应该使用值PCI_ANY_ID。

kernel_ulong_t driver_data;

该值不是用来和设备相匹配的，而是用来保存PCI驱动程序用于区分不同设备的信息，如果它需要的话。

应该使用两个辅助宏来进行struct pci_device_id结构体的初始化：

PCI_DEVICE(vendor, device)

它创建一个仅和特定厂商及设备ID相匹配的struct pci_device_id。这个宏把结构体的subvendor和subdevice字段设置为PCI_ANY_ID。

PCI_DEVICE_CLASS(device_class, device_class_mask)

它创建一个和特定PCI类相匹配的struct pci_device_id。

下面的内核文件中给出了一个使用这些宏来定义驱动程序支持的设备类型的例子:

```
drivers/usb/host/ehci-hcd.c:

static const struct pci_device_id pci_ids[  ] = { {
        /* 由任何USB 2.0 EHCI控制器处理 */
        PCI_DEVICE_CLASS(((PCI_CLASS_SERIAL_USB << 8) | 0x20), ~0),
        .driver_data =  (unsigned long) &ehci_driver,
        },
        { /* 结束: 全部为零 */ }
};

drivers/i2c/busses/i2c-i810.c:
static struct pci_device_id i810_ids[  ] = {
    { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82810_IG1) },
    { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82810_IG3) },
    { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82810E_IG) },
    { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82815_CGC) },
    { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82845G_IG) },
    { 0, },
};
```

这些例子创建了一个struct pci_device_id结构体数组，数组的最后一个值是全部设置为0的空结构体。这个ID数组被用在struct pci_driver中（稍后描述），它还被用于告知用户空间这个特定的驱动程序支持什么设备。

MODULE_DEVICE_TABLE

这个pci_device_id结构体需要被导出到用户空间，使热插拔和模块装载系统知道什么模块针对什么硬件设备。宏MODULE_DEVICE_TABLE完成这个工作。例子:

```
MODULE_DEVICE_TABLE(pci, i810_ids);
```

该语句创建一个名为__mod_pci_device_table的局部变量，指向struct pci_device_id数组。在稍后的内核构建过程中，depmod程序在所有的模块中搜索符号__mod_pci_device_table。如果找到了该符号，它把数据从该模块中抽出，添加到文件*/lib/modules/KERNEL_VERSION/modules.pcimap*中。当depmod结束之后，内核模块支持的所有PCI设备连同它们的模块名都在该文件中被列出。当内核告知热插拔

系统一个新的PCI设备已经被发现时，热插拔系统使用*modules.pcimap*文件来寻找要装载的恰当的驱动程序。

注册 PCI 驱动程序

为了正确地注册到内核，所有的 PCI 驱动程序都必须创建的主要结构体是 `struct pci_driver` 结构体。该结构体由许多回调函数和变量组成，向 PCI 核心描述了 PCI 驱动程序。下面列出了该结构体中 PCI 驱动程序必须注意的字段：

`const char *name;`

驱动程序的名字。在内核的所有PCI驱动程序中它必须是唯一的，通常被设置为和驱动程序的模块名相同的名字。当驱动程序运行在内核中时，它会出现在 sysfs 的 */sys/bus/pci/drivers/* 下面。

`const struct pci_device_id *id_table;`

指向本章前面介绍的 `struct pci_device_id` 表的指针。

`int (*probe) (struct pci_dev *dev, const struct pci_device_id *id);`

指向PCI驱动程序中的探测函数的指针。当PCI核心有一个它认为驱动程序需要控制的 `struct pci_dev` 时，就会调用该函数。PCI 核心用来做判断的 `struct pci_device_id` 指针也被传递给该函数。如果 PCI 驱动程序确认传递给它的 `struct pci_dev`，则应该恰当地初始化设备然后返回 0。如果驱动程序不确认该设备，或者发生了错误，它应该返回一个负的错误值。本章稍后将对该函数做更详细的介绍。

`void (*remove) (struct pci_dev *dev);`

指向一个移除函数的指针，当 `struct pci_dev` 被从系统中移除，或者 PCI 驱动程序正在从内核中卸载时，PCI 核心调用该函数。本章稍后将对该函数做更详细的介绍。

`int (*suspend) (struct pci_dev *dev, u32 state);`

指向一个挂起函数的指针，当 `struct pci_dev` 被挂起时 PCI 核心调用该函数。挂起状态以 `state` 变量来传递。该函数是可选的，驱动程序不一定要提供。

`int (*resume) (struct pci_dev *dev);`

指向一个恢复函数的指针，当 `struct pci_dev` 被恢复时 PCI 核心调用该函数。它总是在挂起函数已经被调用之后被调用。该函数是可选的，驱动程序不一定要提供。

概言之，为了创建一个正确的 `struct pci_driver` 结构体，只需要初始化四个字段：

```
static struct pci_driver pci_driver = {
    .name = "pci_skel",
    .id_table = ids,
    .probe = probe,
    .remove = remove,
};
```

为了把struct pci_driver注册到PCI核心中，需要调用以struct pci_driver指针为参数的*pci_register_driver*函数。通常在PCI驱动程序的模块初始化代码中完成该工作：

```
static int _ _init pci_skel_init(void)
{
    return pci_register_driver(&pci_driver);
}
```

注意，如果注册成功，*pci_register_driver*函数返回0；否则，返回一个负的错误编号。它不会返回绑定到驱动程序的设备的数量，或者在没有设备绑定到驱动程序时返回一个错误编号。这是2.6发布之后的一个变化，基于下列情形的考虑：

- 在支持PCI热插拔的系统或者CardBus系统上，PCI设备可以在任何时刻出现或者消失。如果驱动程序能够在设备出现之前被装载的话是很有帮助的，这样可以减少初始化设备所花的时间。
- 2.6内核允许在驱动程序被装载之后动态地分配新的PCI ID给它。这是通过文件new_id来完成的，该文件位于sysfs的所有PCI驱动程序目录中。这是非常有用的，如果正在使用的新的设备还没有被内核所认知的话。用户可以把PCI ID的值写到*new_id*文件，之后驱动程序就可绑定新的设备。如果在设备没有出现在系统中之前不允许装载驱动程序的话，该接口将不起作用。

当PCI驱动程序将要被卸载的时候，需要把struct pci_driver从内核注销。这是通过调用*pci_unregister_driver*来完成的。当该函数被调用时，当前绑定到该驱动程序的任何PCI设备都被移除，该PCI驱动程序的移除函数在*pci_unregister_driver*函数返回之前被调用。

```
static void _ _exit pci_skel_exit(void)
{
    pci_unregister_driver(&pci_driver);
}
```

老式PCI探测

在稍老的内核版本中，PCI驱动程序并不总是使用*pci_register_driver*函数。它们不是手工搜寻系统中的PCI设备列表，就是调用一个可以查找特定PCI设备的函数。在2.6

内核中，驱动程序搜寻系统中PCI设备列表的能力已经被去掉，以防止驱动程序使内核崩溃；当一个设备正在被移除时，如果驱动程序正好在修改PCI设备列表，就会发生这个危险。

如果真的需要查找特定PCI设备的能力的话，可以使用下面的函数：

```
struct pci_dev *pci_get_device(unsigned int vendor, unsigned int device,
                               struct pci_dev *from);
```

该函数扫描系统中当前存在的PCI设备列表，如果输入参数和指定厂商及设备ID相匹配的话，它增加所发现的struct pci_dev变量的引用计数，然后将其返回给调用者。这避免了该结构体无声地消失，从而保证不出现内核不知所措的情况。当struct pci_dev由该函数返回之后，驱动程序必须调用*pci_dev_put*函数来把使用计数适当地减小，以允许内核在设备被移除时把它清理掉。

from参数用来得到具有同一签名的多个设备；该参数应该指向已经被找到的最近一个设备，那么查找就可以继续而不用从列表头重新开始了。把from指定为NULL可以查找第一个设备。如果没有（其余的）设备被发现，返回NULL。

下面是关于如何正确使用该函数的一个例子：

```
struct pci_dev *dev;
dev = pci_get_device(PCI_VENDOR_FOO, PCI_DEVICE_FOO, NULL);
if (dev) {
    /* 使用PCI驱动程序 */
    ...
    pci_dev_put(dev);
}
```

该函数不能在中断上下文中被调用。如果这么干，会在系统日志中打印一条警告信息。

```
struct pci_dev *pci_get_subsys(unsigned int vendor, unsigned int device,
  unsigned int ss_vendor, unsigned int ss_device, struct pci_dev *from);
```

该函数和*pci_get_device*类似，但它允许在查找设备时指定子系统厂商和子系统设备ID。

该函数不能在中断上下文中被调用。如果这么做，内核会打印一条警告信息到系统日志中。

```
struct pci_dev *pci_get_slot(struct pci_bus *bus, unsigned int devfn);
```

该函数在指定struct pci_bus上的系统PCI设备列表中查找指定的设备和PCI设备的功能编号。如果发现了匹配的设备，该设备的引用计数就会增加，然后返回一个指向它的指针。当调用者结束了对struct pci_dev的使用之后，它必须调用*pci_dev_put*。

所有这些函数都不能在中断上下文中被调用。如果这么干，会在系统日志中打印一条警告信息。

激活 PCI 设备

在 PCI 驱动程序的探测函数中，在驱动程序可以访问 PCI 设备的任何设备资源之前（I/O 区域或者中断），驱动程序必须调用 *pci_enable_device* 函数：

```
int pci_enable_device(struct pci_dev *dev);
```

该函数实际地激活设备。它把设备唤醒，在某些情况下还指派它的中断线和 I/O 区域。CardBus 设备就是这种情况（在驱动程序层和 PCI 完全一样）。

访问配置空间

在驱动程序检测到设备之后，它通常需要读取或写入三个地址空间：内存、端口和配置。对驱动程序而言，对配置空间的访问至关重要，因为这是它找到设备映射到内存和 I/O 空间的什么位置的唯一途径。

因为处理器没有任何直接访问配置空间的途径，因此，计算机厂商必须提供一种办法。为了访问配置空间，CPU 必须读取或写入 PCI 控制器的寄存器，但具体的实现取决于计算机厂商，和我们这里的讨论无关，因为 Linux 提供了访问配置空间的标准接口。

对于驱动程序而言，可通过 8 位、16 位或 32 位的数据传输访问配置空间。相关函数的原型定义在 *<linux/pci.h>* 中：

```
int pci_read_config_byte(struct pci_dev *dev, int where, u8 *val);
int pci_read_config_word(struct pci_dev *dev, int where, u16 *val);
int pci_read_config_dword(struct pci_dev *dev, int where, u32 *val);
```

从由 `dev` 标识的设备配置空间读入一个、两个或四个字节。`where` 参数是从配置空间起始位置计算的字节偏移量。从配置空间获得的值通过 `val` 指针返回，函数本身的返回值是错误码。*word* 和 *dword* 函数会将读取到的 little-endian 值转换成处理器固有的字节序，因此，我们自己无需处理字节序。

```
int pci_write_config_byte(struct pci_dev *dev, int where, u8 val);
int pci_write_config_word(struct pci_dev *dev, int where, u16 val);
int pci_write_config_dword(struct pci_dev *dev, int where, u32 val);
```

向配置空间写入一个、两个或四个字节。和上面的函数一样，`dev` 标识设备，要写入的值通过 `val` 传递。*word* 和 *dword* 函数在把值写入外设之前，会将其转换成小头字节序。

所有前面的函数都实现为inline函数，它们实际上调用下面的函数。在驱动程序不能访问struct pci_dev的任何时刻，都可以使用这些函数来代替上述函数。

```
int pci_bus_read_config_byte (struct pci_bus *bus, unsigned int devfn,
  int where, u8 *val);
int pci_bus_read_config_word (struct pci_bus *bus, unsigned int devfn,
  int where, u16 *val);
int pci_bus_read_config_dword (struct pci_bus *bus, unsigned int devfn,
  int where, u32 *val);
```

类似于*pci_read_*函数，但需用到pci_bus * 和devfn变量，而不用struct pci_dev *。

```
int pci_bus_write_config_byte (struct pci_bus *bus, unsigned int devfn,
  int where, u8 val);
int pci_bus_write_config_word (struct pci_bus *bus, unsigned int devfn,
  int where, u16 val);
int pci_bus_write_config_dword (struct pci_bus *bus, unsigned int devfn,
  int where, u32 val);
```

和*pci_write_*系列函数类似，但是需要struct pci_bus *和devfn变量，而不是struct pci_dev *。

使用*pci_read_*系列函数读取配置变量的首选方法，是使用*<linux/pci.h>*中定义的符号名。例如，下面的小函数通过给*pci_read_config_byte*函数的where参数传递一个符号名来获取设备的修订号ID。

```
static unsigned char skel_get_revision(struct pci_dev *dev)
{
    u8 revision;

    pci_read_config_byte(dev, PCI_REVISION_ID, &revision);
    return revision;
}
```

访问I/O和内存空间

一个PCI设备可实现多达6个I/O地址区域。每个区域可以是内存也可以是I/O地址。大多数设备在内存区域实现I/O寄存器，因为这通常是一个更明智的方法（如第九章的“I/O端口和I/O内存”一节所述）。但是，不像常规内存，I/O寄存器不应该由CPU缓存，因为每次访问都可能有边际作用。将I/O寄存器实现为内存区域的PCI设备通过在其配

置寄存器中设置“内存是可预取的（memory-is-prefetchable）”标志来标记这个不同（注4）。如果内存区域被标记为可预取（prefetchable），则CPU可缓存其内容，并进行各种优化。另一方面，对非可预取的（nonprefetchable）内存的访问不能被优化，因为每个访问都可能有边际作用，就像I/O端口。把控制寄存器映射到内存地址范围的外设把该范围声明为非可预取的，不过像PCI板载视频内存这样的东西是可预取的。在本节中，我们使用“区域”一词来表示一般的I/O地址空间，包括内存映射的和端口映射。

一个接口板通过配置寄存器报告其区域的大小和当前位置——即图12-2中的6个32位寄存器，它们的符号名称为PCI_BASE_ADDRESS_0到PCI_BASE_ADDRESS_5。因为PCI定义的I/O空间是32位地址空间，因此，内存和I/O使用相同的配置接口是有道理的。如果设备使用64位的地址总线，它可以为每个区域使用两个连续的PCI_BASE_ADDRESS寄存器来声明64位内存空间中的区域（低位优先）。对一个设备来说，既提供32位区域也提供64位区域是可能的。

在内核中，PCI设备的I/O区域已经被集成到通用资源管理。因此，我们无需访问配置变量来了解设备被映射到内存或I/O空间的何处。获得区域信息的首选接口由如下函数组成：

```
unsigned long pci_resource_start(struct pci_dev *dev, int bar);
```

该函数返回六个PCI I/O区域之一的首地址（内存地址或I/O端口号）。该区域由整数的bar（base address register，基地址寄存器）指定，bar的取值为0到5。

```
unsigned long pci_resource_end(struct pci_dev *dev, int bar);
```

该函数返回第bar个I/O区域的尾地址。注意这是最后一个可用的地址，而不是该区域之后的第一个地址。

```
unsigned long pci_resource_flags(struct pci_dev *dev, int bar);
```

该函数返回和该资源相关联的标志。

资源标志用来定义单个资源的某些特性。对与PCI I/O区域相关联的PCI资源，该信息从基地址寄存器中获得，但对于和PCI设备无关的资源，它可能来自其他地方。

所有资源标志定义在*<linux/ioport.h>*中；下面列出其中最重要的几个：

```
IORESOURCE_IO
IORESOURCE_MEM
```

如果相关的I/O区域存在，将设置这些标志之一。

注4：该信息保存在PCI寄存器基地址的低位中，这些位定义在*<linux/pci.h>*中。

IORESOURCE_PREFETCH

IORESOURCE_READONLY

这些标志表明内存区域是否为可预取的和/或是写保护的。对PCI资源来说，从来不会设置后面的那个标志。

通过使用*pci_resource_*系列函数、设备驱动程序可完全忽略底层的PCI寄存器，因为系统已经使用这些寄存器构建了资源信息。

PCI 中断

很容易处理PCI的中断。在Linux的引导阶段，计算机固件已经为设备分配了一个唯一的中断号，驱动程序只需使用该中断号。中断号保存在配置寄存器60（PCI_INTERRUPT_LINE）中，该寄存器为一个字节宽。这允许多达256个中断线，但实际的限制取决于所使用的CPU。驱动程序无需检测中断号，因为从PCI_INTERRUPT_LINE中找到的值肯定是正确的。

如果设备不支持中断，寄存器61（PCI_INTERRUPT_PIN）是0；否则为非零。但是，因为驱动程序知道自己的设备是否是中断驱动的，因此，它通常不需要读取PCI_INTERRUPT_PIN寄存器。

这样，处理中断的PCI特定代码仅仅需要读取配置字节，以获取保存在一个局部变量中的中断号，如下面的代码所示。否则，要利用第十章的内容。

```
result = pci_read_config_byte(dev, PCI_INTERRUPT_LINE, &myirq);
if (result) {
    /* 处理错误 */
}
```

本节的剩余内容为好奇的读者提供了一些附加信息，但它们对编写驱动程序没有多少帮助。

PCI连接器有四个中断引脚，外设板可使用其中任意一个或者全部。每个引脚被独立连接到主板的中断控制器，因此，中断可被共享而不会出现任何电气问题。然后，中断控制器负责将中断线（引脚）映射到处理器硬件；这一依赖于平台的操作由控制器来完成，这样，总线本身可以获得平台无关性。

位于PCI_INTERRUPT_PIN的只读配置寄存器用来告诉计算机实际使用的是哪个引脚。要记得每个设备板可容纳最多8个设备；而每个设备使用单独的中断引脚，并在自己的配置寄存器中报告引脚的使用情况。同一设备板上的不同设备可使用不同的中断引脚，或者共享同一个中断引脚。

另一方面，PCI_INTERRUPT_LINE寄存器是可读/写的。在计算机的引导阶段，固件扫描其PCI设备，并根据中断引脚如何连接到它的PCI槽来设置每个设备的寄存器。这个值由固件分配，因为只有固件知道主板如何将不同的中断引脚连接至处理器。但是，对设备驱动程序而言，PCI_INTERRUPT_LINE是只读的。有趣的是，新近的Linux内核在某些情况下无需借助于BIOS就可以分配中断线。

硬件抽象

到此为止，我们通过了解系统如何处理市场上各种各样的PCI控制器，已经完整地讨论了PCI总线。本节只是提供一些资料，以帮助感兴趣的读者了解内核是如何将面向对象的布局扩展至最底层的。

用于实现硬件抽象的机制，就是包含方法的普通结构。这是一种强有力的技术，它只是在普通的函数调用开销之上增加了对指针取值这样一点最小的开销。在PCI管理中，唯一依赖于硬件的操作是读取和写入配置寄存器，因为PCI世界中的任何其他工作，都是通过直接读取和写入I/O及内存地址空间来完成的，而这些工作是由CPU直接控制的。

为此，用于配置寄存器访问的相关结构仅包含2个字段：

```
struct pci_ops {
    int (*read)(struct pci_bus *bus, unsigned int devfn, int where, int size,
                u32 *val);
    int (*write)(struct pci_bus *bus, unsigned int devfn, int where, int size,
                 u32 val);
};
```

该结构在*<linux/pci.h>*中定义，并由*drivers/pci/pci.c*使用，后者定义了实际的公共函数。

作用于PCI配置空间的这两个函数比对指针取值要花费更多的开销；因为代码是高度面向对象的，它们使用了级联指针，但该开销对于执行次数极少而且从来不会在速度要求很高的路径上执行的操作来说并不是一个问题。例如，*pci_read_config_byte(dev, where, val)*的实际实现扩展为：

```
dev->bus->ops->read(bus, devfn, where, 8, val);
```

系统中的各种PCI总线在系统引导阶段得到检测，这时，struct pci_bus项被创建并与其功能关联起来，其中包括ops字段。

通过“硬件操作”数据结构实现硬件抽象在Linux中很典型。一个重要的例子是struct alpha_machine_vector数据结构。该结构体在*<asm-alph/machvec.h>*中定义，并用来处理各种Alpha计算机之间的不同。

ISA 回顾

ISA 总线在设计上相当陈旧而且其差劲的性能臭名昭著，但是，它仍然占有很大一部分的扩展设备市场。当要支持老主板而速度又不是非常重要时，ISA 比起 PCI 要占些优势。ISA 这个老标准的另外一个优点是，如果你是一位电子爱好者，你可以非常容易地设计开发自己的 ISA 设备，而这对 PCI 来说简直是不可能的。

另一方面，ISA 的最大不足在于它被紧紧绑定在 PC 架构上；其接口总线具有 80286 处理器的所有限制，使系统程序员头疼不已。ISA 设计的另外一个大问题（源自最初的 IBM PC）是缺少地理寻址，这导致了许多问题，迫使在添加新设备时要经历漫长的“拔下－重新跳线－插入－测试”周期。值得注意的是，连最老的 Apple II 计算机都采用了地理寻址方法，从而可以装备无跳线的扩展板卡。

尽管 ISA 总线有如此大的缺点，但仍然被应用于若干意想不到的领域。例如，用在几种掌上电脑中的 MIPS 处理器的 VR41xx 系列装备有 ISA 兼容的扩展总线，看起来有点奇怪。这种意想不到的应用的背后原因是某些基于 ISA 的传统硬件的成本极其低廉，例如基于 8390 的以太网卡，这样，利用 ISA 电气信号的 CPU 就能够非常容易地利用这种糟糕但便宜的 PC 设备。

硬件资源

一个 ISA 设备可配备有 I/O 端口、内存区域以及中断线。尽管 x86 处理器支持 64 KB 的 I/O 端口内存（也就是说，处理器有 16 条地址线），但某些老式的 PC 硬件只能处理最低的 10 条地址线。这把可用的地址空间限制在 1024 个端口，因为任何只能处理低位地址线的设备，都会错误地将 1 KB 至 64 KB 范围内的地址看成是低地址。某些外设通过只把一个端口映射到低千字节而使用高地址线来选择不同的设备寄存器来绕过这个限制。例如，映射到 0x340 端口的设备，也可以安全地使用 0x740、0xB40 等端口。

如果可用的 I/O 端口受到限制，内存访问情况就更加糟糕。ISA 设备只能把 640 KB 和 1 MB 之间以及 15 MB 和 16 MB 之间的内存用于 I/O 寄存器和设备控制。640 KB 到 1 MB 的范围由 PC BIOS、VGA 兼容适配器，以及其他各种设备使用，新设备能用的空间非常有限。另一方面，Linux 不直接支持 15 MB 处的内存访问，如果想通过修改内核来支持它现在已经得不偿失了。

ISA 设备板可利用的第三个资源是中断线。连接到 ISA 总线的中断线非常有限，而且由所有的接口板卡共享。这样，如果设备配置不当，将出现多个不同设备使用同一中断线的结果。

尽管最初的ISA规范不允许在设备间共享中断，但大多数设备板都允许（注5）。软件级别的中断共享在第十章的“中断共享”一节中讲述。

ISA编程

对编程而言，内核或BIOS都没有提供任何特定的帮助来使访问ISA设备更加容易（和PCI不同）。我们能利用的唯一设施是I/O端口寄存器以及IRQ线，相关论述可参阅第十章的“安装中断处理程序”一节。

本书第一部分中讲述的所有编程技术都可以应用于ISA设备；驱动程序可以探测I/O端口，而中断线的自动检测必须利用第十章的“自动检测IRQ号”一节中描述的技术之一。

辅助函数 *isa_readb* 以及其他相关函数已经在第十章的“使用I/O内存”中简要介绍过了，这里不再赘述。

即插即用规范

某些新的ISA设备板遵循特殊的设计规则，需要一个特殊的初始化序列，以便简化附加接口板的安装和配置。这些接口板的设计规范被称为PnP（Plug and Play，即插即用），其中包括一堆建立和配置无跳线ISA设备的麻烦规则。PnP设备实现了可重分配的I/O区域；而PC BIOS负责重新分配（请回忆PCI的相关内容）。

简而言之，PnP的目标就是获得类似PCI设备那样的灵活性，而无需修改底层的电气接口（即ISA总线）。为此，该规范定义了一组设备无关的配置寄存器，以及地理寻址接口板的方法——虽然物理总线并不支持各板独立的（地理）连线，而每个ISA信号线都会连接到每个插槽。

地理寻址通过为计算机中的每个PnP外设分配一个小整数来工作，称为CSN（Card Select Number，卡选择号）。每个PnP设备配备有一个唯一的序列标识号，有64位宽，并且被硬编码到外设板中。CSN分配利用该唯一序列号来识别PnP设备。但是，只能在引导阶段对CSN进行安全的分配，而这需要BIOS能够识别PnP设备。出于这个原因，老的计算机需要用户获得并插入一张特殊的配置磁盘，即使设备具有PnP功能也不例外。

注5：和中断共享相关的问题可通过电子工程来解释：如果设备驱动程序驱动信号线为不活动状态（通过发出低阻抗电平信号），则中断就无法共享。另一方面，如果设备使用拉升电阻导致不活动的逻辑电平，则共享就有可能。这已经成为目前的标准。但是，仍然存在丢失中断事件的潜在风险，因为ISA中断是边缘触发的而不是电平触发的。边缘触发中断在硬件上更加容易实现，但对共享来说并不安全。

遵循 PnP 规范的接口板在硬件层次上比较复杂。与 PCI 板相比，它们更为精细，而且要求更为复杂的软件。安装这些设备时一样会遇到麻烦，即使安装很顺利，也仍然要面对性能限制以及有限的 ISA 总线 I/O 空间等问题。因此，只要可能，应该尽量安装 PCI 设备，体验新技术。

如果读者对 PnP 配置软件感兴趣，可浏览 *drivers/net/3c509.c*，这个驱动程序的探测函数处理 PnP 设备。2.6 内核在 PnP 设备的支持方面做了很多工作，因此，和前一次内核发布相比，许多不灵活的接口已经被清理掉了。

PC/104 和 PC/104+

在工业界，当前有两种非常流行的总线架构：PC/104 和 PC/104+。它们都是 PC 类单板计算机的标准。

这两个标准都规定了印刷电路板的外形，以及板间互连的电气 / 机械规范。这些总线的实际好处在于，它们可以使用在设备一面的插头 - 插座类型的连接器把多个电路板垂直堆叠起来。

这两个总线的电子和逻辑布局分别和 ISA（PC/104）及 PCI（PC/104+）一样，因此，软件不会注意到它们和通常桌面总线之间的不同。

其他的 PC 总线

PCI 和 ISA 是 PC 领域最常用的外设接口，但它们并不是仅有的 PC 总线。这里给出了能在 PC 市场上找到的其他一些总线。

MCA

MCA（Micro Channel Architecture，微通道结构）是在 PS/2 计算机和某些笔记本电脑中使用的 IBM 标准。在硬件层次上，微通道的功能比 ISA 更多。它支持多主（multimaster）DMA、32 位地址和数据线、共享中断线和用来访问板载配置寄存器的地理寻址等等。这种寄存器被称为 POS（可编程选项选择，*Programmable Option Select*），但它们不具有 PCI 寄存器的所有功能。Linux 对微通道的支持包括导出给模块使用的函数。

设备驱动程序可以读取整数值 `MCA_bus`，以便判断是否运行在微通道计算机上。如果该符号是一个预处理宏，那么 `MCA_bus__is_a_macro` 宏也会被定义。如果 `MCA_bus__is_a_macro` 未被定义，则 `MCA_bus` 是一个导出到模块化代码的整型变量。`MCA_bus` 和 `MCA_bus__is_a_macro` 在 *<asm/processor.h>* 中定义。

EISA

扩展ISA（EISA）总线是对ISA总线的32扩展，同时具有兼容的接口连接器；ISA设备板可以插入ISA连接器。附加的线路在ISA接点之下走线。

类似PCI和MCA，EISA总线也是为无跳线设备而设计的，并具有和MCA一样的特点：32位地址和数据线、多主DMA和共享中断线。EISA设备由软件配置，但它们不需要操作系统的任何特殊支持。Linux内核中已经有一些EISA驱动程序，包括以太网设备和SCSI控制器。

EISA驱动程序检查`EISA_bus`的值来判断主机是否装备有EISA总线。和`MCA_bus`类似，`EISA_bus`可以是宏，也可以是变量，取决于`EISA_bus__is_a_macro`是否被定义。这两个符号均定义在*<asm/processor.h>*中。

对于有sysfs和资源管理功能的设备内核提供了完全的EISA支持，位于*drivers/eisa*目录。

VLB

另外一个对ISA的扩展是VLB（VESA Local Bus，VESA局部总线）接口总线，它通过添加第三个纵向插槽对ISA连接器进行了扩展。设备可以插入这个额外的连接器（不需要插入另外两个相关的ISA连接器插槽），因为VLB插槽复制了ISA连接器中的所有重要信号。这种不使用ISA槽的“独立”的VLB外设很少见，因为大多数设备需要接触到计算机的背板才能连接到外部连接器上。

与EISA、MCA和PCI总线相比，VESA总线的功能更加有限，因此正在从市场上消失。内核中不存在对VLB的任何特殊支持。但是，Linux 2.0中的Lance以太网驱动程序和IDE磁盘驱动程序可处理这些设备的VLB版本。

SBus

在现今大部分计算机装备PCI或ISA接口总线的同时，大部分稍老的SPARC工作站使用SBus连接它们的外设。

尽管SBus存在很长一段时间了，但它具有相当高级的设计。尽管只有SPARC计算机使用该总线，但它的初衷却是处理器无关的，并针对I/O外设板进行了优化。换句话说，我们可以将额外的RAM插入SBus插槽（RAM扩展板已经从ISA领域消失，而PCI根本不支持它们）。这种优化可简化硬件设备和系统软件的设计，其代价是主板更加复杂一些。

SBus总线的这种I/O处理方法导致外设使用“虚拟”地址来传输数据，以绕过分配连续DMA缓冲区的需求。主板负责将虚拟地址解码并映射到物理地址。这要求在总线上附加一个MMU（内存管理单元）；负责该任务的芯片被称为IOMMU。与使用物理地址的接口总线相比，这种设计似乎有些复杂，但因为SPARC处理器始终将MMU核心从CPU核心中分离（要么是在物理上，要么至少是在概念上），从而使之大大简化。实际上，这种设计决策也被其他巧妙的处理器设计共享，从而获得整体上的好处。这种总线的另外一个好处是，设备板使用大量的地理寻址，因此不需要在每一个外设中实现地址解码器或者处理地址冲突。

SBus外设在它们的PROM中使用Forth语言来初始化它们自身。选择Forth的原因是，其解释器是轻量级的，因此可以在任何计算机系统的固件中实现。此外，SBus规范描述了引导过程，因此，兼容的I/O设备能很容易地融合到系统中，并且在系统引导时被识别出来。对支持多平台的设备来说，这一步意义非凡；这完全不同于惯常的以PC为中心的ISA领域。但是，由于许多商业原因，这个总线并未取得成功。

尽管当前内核版本对SBus设备提供了相当完善的支持，但该总线已经很少被用到，因此不值得在这里详述。感兴趣的读者可以查看*arch/sparc/kernel*和*arch/sparc/mm*中的源文件。

NuBus

另外一个有趣但几乎被遗忘的接口总线是NuBus。你可以在老式的Mac计算机（使用M68k系列CPU）中找到它。

所有的总线都是内存映射的（类似M68k中的所有东西），而且设备只能被地理寻址。这是Apple的优点和风格，因为更老式的Apple II都已经具备类似的总线布局。其缺陷在于，几乎不太可能找到任何有关NuBus的文档，这归咎于Apple在Mac计算机上一贯遵循的封闭一切的策略（不同于先前的Apple II系统，其源代码和图表可以花非常低的代价获得）。

文件*drivers/nubus/nubus.c*包含了我们就该总线所知道的一切，读起来也相当有趣；从中可以看出，开发人员不得不做了多少艰难的逆向工程。

外部总线

接口总线领域最近出现了一个新的家族：外部总线。这包括USB、FireWire和IEEE1284（基于并口的外部总线）。这些接口在某种程度上和老式的、非外部的技术（例如PCMCIA/CardBus），甚至SCSI类似。

从概念上讲，这些总线既不是功能完整的接口总线（比如PCI），也不是哑的通信通道（比如串口）。很难对利用其功能的软件进行分类，通常可划分为两个级别：硬件控制器的驱动程序（比如针对PCI SCSI适配器的驱动程序，或者在“PCI接口”一节中描述过的PCI控制器），以及针对特定“客户”设备的驱动程序（比如处理通用SCSI磁盘的*sd.c*和处理插入总线的板卡的PCI驱动程序）。

快速参考

本节总结本章中介绍过的符号：

```
#include <linux/pci.h>
```

这个头文件包含PCI寄存器的符号名称，以及若干厂商和设备ID值。

```
struct pci_dev;
```

代表内核中PCI设备的结构体。

```
struct pci_driver;
```

代表PCI驱动程序的结构体。所有的PCI驱动程序必须定义该结构体。

```
struct pci_device_id;
```

描述该驱动程序所支持的PCI设备类型的结构体。

```
int pci_register_driver(struct pci_driver *drv);
int pci_module_init(struct pci_driver *drv);
void pci_unregister_driver(struct pci_driver *drv);
```

从内核注册或者注销PCI驱动程序的函数。

```
struct pci_dev *pci_find_device(unsigned int vendor, unsigned int device,
                                struct pci_dev *from);
struct pci_dev *pci_find_device_reverse(unsigned int vendor, unsigned int
                                device, const struct pci_dev *from);
struct pci_dev *pci_find_subsys (unsigned int vendor, unsigned int device,
  unsigned int ss_vendor, unsigned int ss_device, const struct pci_dev *from);
struct pci_dev *pci_find_class(unsigned int class, struct pci_dev *from);
```

在设备列表中查找具有特定签名或者属于某一特定类的设备的函数。如果没有找到，返回值为NULL。from被用来继续查找；在第一次调用函数时它必须为NULL，如果想要查找更多的设备，它必须指向前一个找到的设备。这些函数不建议使用，应该用pci_get_系列函数代替。

```
struct pci_dev *pci_get_device(unsigned int vendor, unsigned int device,
                               struct pci_dev *from);
struct pci_dev *pci_get_subsys(unsigned int vendor, unsigned int device,
  unsigned int ss_vendor, unsigned int ss_device, struct pci_dev *from);
struct pci_dev *pci_get_slot(struct pci_bus *bus, unsigned int devfn);
```

在设备列表中查找具有特定签名或者属于某一特定类的设备的函数。如果没有找到，返回值为NULL。from被用来继续查找；在第一次调用函数时它必须为NULL，如果想要查找更多的设备，它必须指向前一个找到的设备。返回的结构体的引用计数被增加，在使用完该结构体之后，必须调用pci_dev_put函数。

```
int pci_read_config_byte(struct pci_dev *dev, int where, u8 *val);
int pci_read_config_word(struct pci_dev *dev, int where, u16 *val);
int pci_read_config_dword(struct pci_dev *dev, int where, u32 *val);
int pci_write_config_byte (struct pci_dev *dev, int where, u8 *val);
int pci_write_config_word (struct pci_dev *dev, int where, u16 *val);
int pci_write_config_dword (struct pci_dev *dev, int where, u32 *val);
```

读取或者写入PCI配置寄存器的函数。尽管Linux内核处理了字节序问题，但从单字节装配多字节值时，程序员必须小心处理字节序问题。PCI总线是小头的。

```
int pci_enable_device(struct pci_dev *dev);
```

激活一个 PCI 设备。

```
unsigned long pci_resource_start(struct pci_dev *dev, int bar);
unsigned long pci_resource_end(struct pci_dev *dev, int bar);
unsigned long pci_resource_flags(struct pci_dev *dev, int bar);
```

处理 PCI 设备资源的函数。

第十三章

USB 驱动程序



通用串行总线（USB）是主机和外围设备之间的一种连接。USB最初是为了替代许多不同的低速总线（包括并行、串行和键盘连接）而设计的，它以单一类型的总线连接各种不同类型的设备（注1）。USB的发展已经超越了这些低速的连接方式，它现在可以支持几乎所有可以连接到PC上的设备。最新的USB规范修订增加了理论上高达480 Mbps的高速连接。

从拓扑上来看，一个USB子系统并不是以总线的方式来布置的；它是一棵由几个点对点的连接构建而成的树。这些连接是连接设备和集线器（hub）的四线电缆（地线、电源线和两根信号线），这和以太网双绞线类似。USB主控制器（host controller）负责询问每一个USB设备是否有数据需要发送。因为这种拓扑布局的原因，一个USB设备在没有主控制器要求的情况下是不能发送数据的。这种配置便于搭建一个非常简易的即插即用类型的系统，藉此，设备可以由主机自动地配置。

USB总线在技术层面上是非常简单的，因为它是一个单主方式的实现，在此方式下，主机轮询各种不同的外围设备。尽管存在这种内在的局限性，USB总线有一些吸引人的特性，例如设备具有要求一个固定的数据传输带宽的能力，以可靠地支持视频和音频I/O。USB另一个重要的特性是它只担当设备和主控制器之间通信通道的角色，对它所发送的数据没有任何特殊的内容和结构上的要求（注2）。

USB协议规范定义了一套任何特定类型的设备都可以遵循的标准。如果一个设备遵循该标准，就不需要一个特殊的驱动程序。这些不同的特定类型称为类（class），包括存储

注1：本章部分内容基于Linux内核USB代码的内核文档，这些文档由内核的USB开发者编写，并且按照GPL条款发布。

注2：实际上，还是存在一些结构，但通常被降低为满足某几个预定义类之一的通信需求：例如，键盘不需要分配带宽，而某些摄像头需要。

设备、键盘、鼠标、游戏杆、网络设备和调制解调器。对于不符合这些类的其他类型的设备，需要针对特定的设备编写一个特定于供货商的驱动程序。视频设备和USB到串口转换设备是一个很好的例子，对于它们没有已定义的标准，来自不同制造商的每一种不同的设备都需要对应的驱动程序。

这些特性，加上设计上与生俱来的热插拔能力，使得USB成为一个便利和低成本的机制，它可以连接多个设备到计算机，而不需要关闭系统、打开机箱、拧螺丝钉和插拔电线。

Linux内核支持两种主要类型的USB驱动程序：宿主（host）系统上的驱动程序和设备（device）上的驱动程序。从宿主的观点来看（一个普通的USB宿主是一个桌面计算机），宿主系统的USB驱动程序控制插入其中的USB设备，而USB设备的驱动程序控制该设备如何作为一个USB设备和主机通信。由于术语“USB设备驱动程序”（USB device drivers）非常易于混淆，USB开发者创建了术语“USB器件驱动程序”（USB gadget drivers）来描述控制连接到计算机（不要忘了Linux还运行于很多小型嵌入式设备上）的USB设备的驱动程序。本章将详细介绍运行于桌面计算机上的USB系统是如何运作的。USB器件驱动程序此刻还未列入本书的内容范围。

如图13-1所示，USB驱动程序存在于不同的内核子系统（块设备、网络设备、字符设备等等）和USB硬件控制器之中。USB核心为USB驱动程序提供了一个用于访问和控制USB硬件的接口，而不必考虑系统当前存在的各种不同类型的USB硬件控制器。

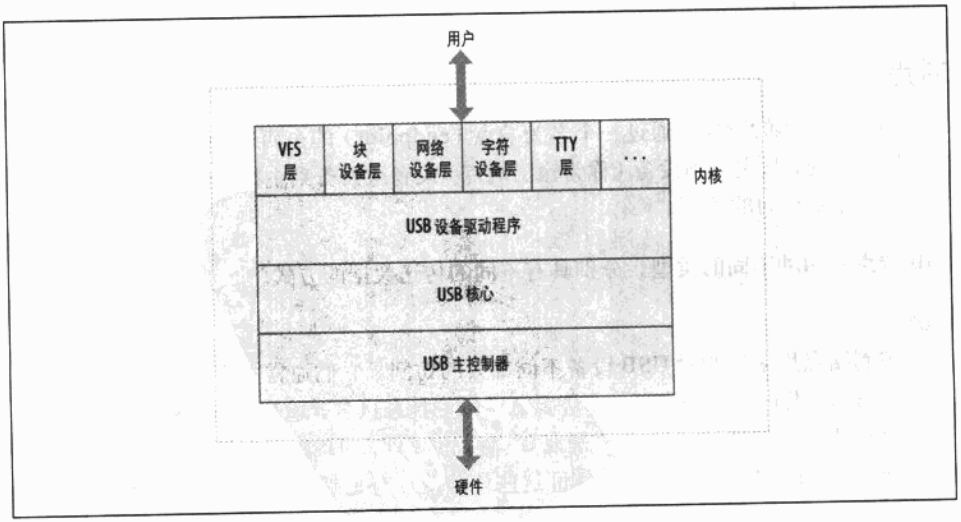


图13-1：USB驱动程序概观

USB 设备基础

USB 设备是一个非常复杂的东西，官方 USB 文档（可由 *http://www.usb.org* 获取）中有详细的描述。幸运的是，Linux 内核提供了一个称为 USB 核心（USB core）的子系统来处理大部分的复杂性。本章描述驱动程序和 USB 核心之间的接口。图 13-2 展示了 USB 设备的构成，包括配置、接口和端点，以及 USB 驱动程序如何绑定到 USB 接口上，而不是整个 USB 设备。

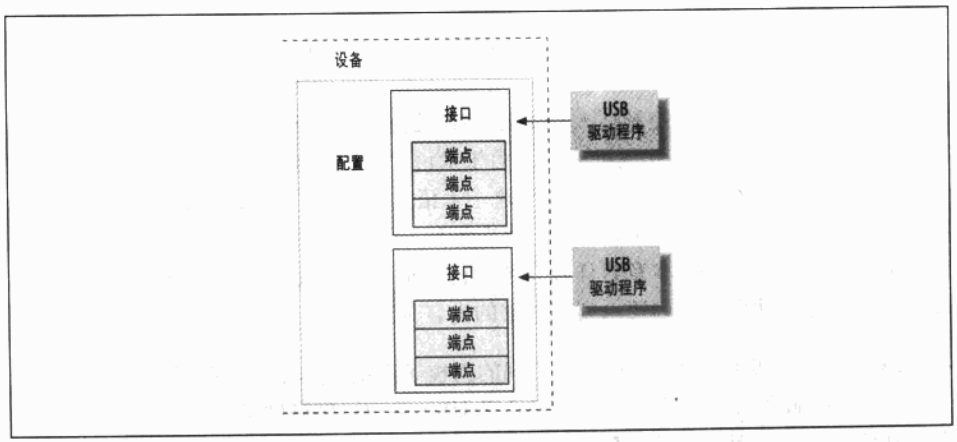


图 13-2：USB 设备概观

端点

USB 通信最基本的形式是通过一个名为端点（endpoint）的东西。USB 端点只能往一个方向传送数据，从主机到设备（称为输出端点）或者从设备到主机（称为输入端点）。端点可以看作是单向的管道。

USB 端点有四种不同的类型，分别具有不同的传送数据的方式：

控制

控制端点用来控制对 USB 设备不同部分的访问。它们通常用于配置设备、获取设备信息、发送命令到设备，或者获取设备的状态报告。这些端点一般体积较小。每个 USB 设备都有一个名为“端点 0”的控制端点，USB 核心使用该端点在插入时进行设备的配置。USB 协议保证这些传输始终有足够的保留带宽以传送数据到设备。

中断

每当USB宿主要求设备传输数据时，中断端点就以一个固定的速率来传送少量的数据。这些端点是USB键盘和鼠标所使用的主要传输方式。它们通常还用于发送数据到USB设备以控制设备，不过一般不用来传输大量的数据。USB协议保证这些传输始终有足够的保留带宽以传送数据。

批量

批量（bulk）端点传输大批量的数据。这些端点通常比中断端点大得多（它们可以一次持有更多的字符）。它们常见于需要确保没有数据丢失的传输的设备。USB协议不保证这些传输始终可以在特定的时间内完成。如果总线上的空间不足以发送整个批量包，它将被分割为多个包进行传输。这些端点通常出现在打印机、存储设备和网络设备上。

等时

等时（isochronous）端点同样可以传送大批量的数据，但数据是否到达是没有保证的。这些端点用于可以应付数据丢失情况的设备，这类设备更注重于保持一个恒定的数据流。实时的数据收集（例如音频和视频设备）几乎毫无例外都使用这类端点。

控制和批量端点用于异步的数据传输，只要驱动程序决定使用它们。中断和等时端点是周期性的。也就是说，这些端点被设置为在固定的时段连续地传输数据，基于此，USB核心为它们保留了相应的带宽。

内核中使用struct usb_host_endpoint结构体来描述USB端点。该结构体在另一个名为struct usb_endpoint_descriptor的结构体中包含了真正的端点信息。后一个结构体包含了所有的USB特定的数据，这些数据的格式是由设备自己定义的。该结构体中驱动程序需要关心的字段有：

`bEndpointAddress`

这是特定端点的USB地址。这个8位的值中还包含了端点的方向。该字段可以结合位掩码USB_DIR_OUT和USB_DIR_IN来使用，以确定该端点的数据是传向设备还是主机。

`bmAttributes`

这是端点的类型。该值可以结合位掩码USB_ENDPOINT_XFERTYPE_MASK来使用，以确定此端点的类型是USB_ENDPOINT_XFER_ISOC、USB_ENDPOINT_XFER_BULK还是USB_ENDPOINT_XFER_INT。这些宏分别表示等时、批量和中断端点。

`wMaxPacketSize`

这是该端点一次可以处理的最大字节数。注意，驱动程序可以发送数量大于此值的

数据到端点，但是在实际传输到设备的时候，数据将被分割为wMaxPacketSize大小的块。对于高速设备，通过使用高位中一些额外的位，该字段可以用来支持端点的高带宽模式。请参考USB规范以了解具体实现的详情。

bInterval

如果端点是中断类型，该值是端点的间隔设置——也就是说，端点的中断请求间隔时间。该值以毫秒为单位。

该结构体的字段并没有采用“传统的”Linux内核命名方案。这是因为这些字段直接对应于USB规范中的字段名字。USB内核程序员认为使用规范指定的名字比使用Linux程序员熟悉的变量命名方式更加重要，因为这样便于规范的阅读。

接口

USB端点被捆绑为接口。USB接口只处理一种USB逻辑连接，例如鼠标、键盘或者音频流。一些USB设备具有多个接口，例如USB扬声器可以包括两个接口：一个USB键盘用于按键和一个USB音频流。因为一个USB接口代表了一个基本功能，而每个USB驱动程序控制一个接口，因此，以扬声器为例，Linux需要两个不同的驱动程序来处理一个硬件设备。

USB接口可以有其他的设置，这些是和接口的参数不同的选择。接口的最初状态是在第一个设置，编号为0。其他的设置可以用来以不同的方式控制端点，例如为设备保留大小不同的USB带宽。每个带有等时端点的设备对同一个接口使用不同的设置。

内核使用struct usb_interface结构体来描述USB接口。USB核心把该结构体传递给USB驱动程序，之后由USB驱动程序来负责控制该结构体。该结构体中的重要字段有：

struct usb_host_interface *altsetting

一个接口结构体数组，包含了所有可能用于该接口的可选设置。每个struct usb_host_interface结构体包含一套由上述struct usb_host_endpoint结构体定义的端点配置。注意，这些接口结构体没有特定的次序。

unsigned num_altsetting

altsetting指针所指的可选设置的数量。

struct usb_host_interface *cur_altsetting

指向altsetting数组内部的指针，表示该接口的当前活动设置。

int minor

如果捆绑到该接口的USB驱动程序使用USB主设备号，这个变量包含USB核心分配给该接口的次设备号。这仅在一个成功的usb_register_dev调用之后才有效（在本章稍后描述）。

struct usb_interface结构体中还有其他的字段，不过USB驱动程序不需要考虑它们。

配置

USB接口本身被捆绑为配置。一个USB设备可以有多个配置，而且可以在配置之间切换以改变设备的状态。例如，一些允许下载固件到其上的设备包含多个配置以完成这个工作，而一个时刻只能激活一个配置。Linux对多个配置的USB设备处理得不是很好，不过，幸好这种情况很少发生。

Linux使用struct usb_host_config结构体来描述USB配置，使用struct usb_device结构体来描述整个USB设备。USB设备驱动程序通常不需要读取或者写入这些结构体中的任何值，因此这里就不详述它们了。想要深入探究的读者可以在内核源代码树的*include/linux/usb.h*文件中找到对它们的描述。

USB设备驱动程序通常需要把一个给定的struct usb_interface结构体的数据转换为一个struct usb_device结构体，USB核心在很多函数调用中都需要该结构体。interface_to_usbdev就是用于该转换功能的函数。

可以期待的是，当前需要struct usb_device结构体的所有USB调用将来会变为使用一个struct usb_interface参数，而且驱动程序不再需要去做转换的工作。

概言之，USB设备是非常复杂的，它由许多不同的逻辑单元组成。这些逻辑单元之间的关系可以简单地描述如下：

- 设备通常具有一个或者更多的配置
- 配置经常具有一个或者更多的接口
- 接口通常具有一个或者更多的设置
- 接口没有或者具有一个以上的端点

USB 和 Sysfs

由于单个USB物理设备的复杂性，在sysfs中表示该设备也相当复杂。无论是物理USB

设备（用struct usb_device表示）还是单独的USB接口（用struct usb_interface表示），在sysfs中均表示为单独的设备（这是因为这些结构体都包含一个struct device结构体）。以仅包含一个USB接口的简易USB鼠标为例，下面是该设备的sysfs目录树：

```
/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1
|-- 2-1:1.0
|   |-- bAlternateSetting
|   |-- bInterfaceClass
|   |-- bInterfaceNumber
|   |-- bInterfaceProtocol
|   |-- bInterfaceSubClass
|   |-- bNumEndpoints
|   |-- detach_state
|   |-- iInterface
|   `-- power
|       `-- state
|-- bConfigurationValue
|-- bDeviceClass
|-- bDeviceProtocol
|-- bDeviceSubClass
|-- bMaxPower
|-- bNumConfigurations
|-- bNumInterfaces
|-- bcdDevice
|-- bmAttributes
|-- detach_state
|-- devnum
|-- idProduct
|-- idVendor
|-- maxchild
|-- power
|   `-- state
|-- speed
`-- version
```

struct usb_device表示为目录树中的：

```
/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1
```

而鼠标的USB接口（USB鼠标驱动程序所绑定的接口）位于如下目录：

```
/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1/2-1:1.0
```

我们将描述内核如何分类USB设备，以帮助理解上面这些长长的设备路径名的含义。

第一个USB设备是一个根集线器（root hub）。这是一个USB控制器，通常包含在一个PCI设备中。之所以这样命名该控制器，是因为它控制着连接到其上的整个USB总线。该控制器是连接PCI总线和USB总线的桥，也是该总线上的第一个USB设备。

所有的根集线器都由USB核心分配了一个独特的编号。在我们的例子中，根集线器名为usb2，因为它是注册到USB核心的第二个根集线器。单个系统中可以包含的根集线器的编号在任何时候都是没有限制的。

USB总线上的每个设备都以根集线器的编号作为其名字中的第一个号码。该号码随后是一个横杠字符和设备所插入的端口号。因为我们例子中的设备插入到第一个端口，1被添加到了名字中。因此，主USB鼠标设备的设备名是2-1。因为该USB设备包含一个接口，导致了树中的另一个设备被添加到sysfs路径中。USB接口的命名方案是设备名直到该接口为止：在我们的例子中，是2-1后面加一个冒号和USB配置的编号，然后是一个句点和接口的编号。因此对于本例而言，设备名是2-1:1.0，因为它是第一个配置，具有接口编号零。

概言之，USB sysfs设备命名方案为：

```
根集线器-集线器端口号:配置.接口
```

随着设备更深地进入USB树，和越来越多的USB集线器的使用，集线器的端口号被添加到跟随着链中前一个集线器端口号的字符串中。对于一个两层的树，其设备名类似于：

```
根集线器-集线器端口号-集线器端口号:配置.接口
```

从前面的USB设备和接口的目录列表可以看到，所有的USB特定信息都可以从sysfs直接获得（例如，idVendor、idProduct和bMaxPower信息）。这些文件中的一个，即*bConfigurationValue*，可以被写入以改变当前使用的活动USB配置。当内核不能够确定选择哪一个配置以恰当地操作设备时，这对于具有多个配置的设备很有用。许多USB调制解调器需要向该文件中写入适当的配置值，以便把恰当的USB驱动程序绑定到该设备。

sysfs并没有展示USB设备所有的不同部分，它只限于接口级别。设备可能包含的任何可选配置都没有显示，还有和接口相关联的端点的细节。这个信息可以从*usbfs*文件系统找到，该文件系统被挂装到系统的*/proc/bus/usb/*目录。*/proc/bus/usb/devices*文件确实显示了和sysfs所展示的所有信息相同的信息，还有系统中存在的所有USB设备的可选配置和端点信息。*usbfs*还允许用户空间的程序直接访问USB设备，这使得许多内核驱动程序可以迁移到用户空间，从而更加容易维护和调试。USB扫描仪是一个很好的例子，它不再存在于内核中，因为它的功能现在包含在了用户空间的SANE库程序中。

USB urb

Linux内核中的USB代码通过一个称为*urb*（USB请求块）的东西和所有的USB设备通信。这个请求块使用struct urb结构体来描述，可以从*include/linux/usb.h*文件中找到。

urb被用来以一种异步的方式往/从特定的USB设备上的特定USB端点发送/接收数据。它的使用和文件系统异步I/O代码中的kiocb结构体以及网络代码中的struct skbuff很类似。USB设备驱动程序可能会为单个端点分配许多urb，也可能对许多不同的端点重用单个的urb，这取决于驱动程序的需要。设备中的每个端点都可以处理一个urb队列，所以多个urb可以在队列为空之前发送到同一个端点。一个urb的典型生命周期如下：

- 由USB设备驱动程序创建。
- 分配给一个特定USB设备的特定端点。
- 由USB设备驱动程序递交到USB核心。
- 由USB核心递交到特定设备的特定USB主控制器驱动程序。
- 由USB主控制器驱动程序处理，它从设备进行USB传送。
- 当urb结束之后，USB主控制器驱动程序通知USB设备驱动程序。

urb可以在任何时刻被递交该urb的驱动程序取消掉，或者被USB核心取消，如果该设备已从系统中移除。urb被动态地创建，它包含一个内部引用计数，使得它们可以在最后一个使用者释放它们时自动地销毁。

本章描述的处理urb的过程是很有用的，因为它使得流处理和其他复杂的、重叠的通信成为可能，而这使驱动程序可以获得最高可能的数据传输速度。不过如果只是想要发送单独的数据块或者控制消息，而不关心数据的吞吐率，过程就不必如此繁琐。（请参考“不使用urb的USB传输”一节）。

struct urb

struct urb结构体中USB设备驱动程序需要关心的字段有：

struct usb_device *dev

urb所发送的目标struct usb_device指针。该变量在urb可以被发送到USB核心之前必须由USB驱动程序初始化。

unsigned int pipe

urb所要发送的特定目标struct usb_device的端点信息。该变量在urb可以被发送到USB核心之前必须由USB驱动程序初始化。

驱动程序必须使用下列恰当的函数来设置该结构体的字段，具体取决于传输的方向。注意每个端点只能属于一种类型。

unsigned int usb_sndctrlpipe(struct usb_device *dev, unsigned int endpoint)

把指定 USB 设备的指定端点号设置为一个控制 OUT 端点。

unsigned int usb_rcvctrlpipe(struct usb_device *dev, unsigned int endpoint)

把指定 USB 设备的指定端点号设置为一个控制 IN 端点。

unsigned int usb_sndbulkpipe(struct usb_device *dev, unsigned int endpoint)

把指定 USB 设备的指定端点号设置为一个批量 OUT 端点。

unsigned int usb_rcvbulkpipe(struct usb_device *dev, unsigned int endpoint)

把指定 USB 设备的指定端点号设置为一个批量 IN 端点。

unsigned int usb_sndintpipe(struct usb_device *dev, unsigned int endpoint)

把指定 USB 设备的指定端点号设置为一个中断 OUT 端点。

unsigned int usb_rcvintpipe(struct usb_device *dev, unsigned int endpoint)

把指定 USB 设备的指定端点号设置为一个中断 IN 端点。

unsigned int usb_sndisocpipe(struct usb_device *dev, unsigned int endpoint)

把指定 USB 设备的指定端点号设置为一个等时 OUT 端点。

unsigned int usb_rcvisocpipe(struct usb_device *dev, unsigned int endpoint)

把指定 USB 设备的指定端点号设置为一个等时 IN 端点。

unsigned int transfer_flags

该变量可以被设置为许多不同的位值，取决于USB驱动程序对urb的具体操作。可用的值包括:

URB_SHORT_NOT_OK

如果被设置，该值说明任何可能发生的对 IN 端点的简短读取应该被 USB 核心当作是一个错误。该值只对从 USB 设备读取的 urb 有用，对用于写入的 urb 没意义。

URB_ISO_ASAP

如果该 urb 是等时的，当驱动程序想要该 urb 被调度时可以设置这个位，只要

带宽的利用允许它这么做，而且想要在此时设置 urb 中的 start_frame 变量。如果一个等时的 urb 没有设置该位，驱动程序必须指定 start_frame 的值，如果传输在当时不能启动的话必须能够正确地恢复。详情请参阅下一节的等时 urb 部分。

URB_NO_TRANSFER_DMA_MAP

当 urb 包含一个即将传输的 DMA 缓冲区时应该设置该位。USB 核心使用 transfer_dma 变量所指向的缓冲区，而不是 transfer_buffer 变量所指向的。

URB_NO_SETUP_DMA_MAP

和 URB_NO_TRANSFER_DMA_MAP 位类似，该位用于控制带有已设置好的 DMA 缓冲区的 urb。如果它被设置，USB 核心使用 setup_dma 变量所指向的缓冲区，而不是 setup_packet 变量。

URB_ASYNC_UNLINK

如果被设置，对该 urb 的 *usb_unlink_urb* 调用几乎立即返回，该 urb 的链接在后台被解开。否则，此函数一直等到 urb 被完全解开链接和结束才返回。使用该位时要小心，因为它可能会造成非常难以调试的同步问题。

URB_NO_FSBR

仅由 UHCI USB 主控制器驱动程序使用，指示它不要企图使用前端总线回收（Front Side Bus Reclamation）逻辑。该位通常不应该被设置，因为带有 UHCI 主控制器的机器会导致大量的 CPU 负荷，而 PCI 总线忙于等待一个设置了该位的 urb。

URB_ZERO_PACKET

如果被设置，一个批量输出 urb 以发送一个不包含数据的小数据包来结束，这时数据对齐到一个端点数据包边界。一些断线的 USB 设备（例如许多 USB 到 IR 设备）需要该位才能正确地工作。

URB_NO_INTERRUPT

如果被设置，当 urb 结束时，硬件可能不会产生一个中断。对该位的使用应当小心谨慎，只有把多个 urb 排队到同一个端点时才使用。USB 核心的函数使用该位来进行 DMA 缓冲区传输。

void *transfer_buffer

指向用于发送数据到设备（OUT urb）或者从设备接收数据（IN urb）的缓冲区的指针。为了使主控制器正确地访问该缓冲区，必须使用 kmalloc 来创建它，而不是在栈中或者静态内存中。对于控制端点，该缓冲区用于传输数据的中转。

dma_addr_t transfer_dma

用于以 DMA 方式传输数据到 USB 设备的缓冲区。

int transfer_buffer_length

transfer_buffer或者transfer_dma变量所指向的缓冲区的大小（因为一个urb只能使用其中一个）。如果该值为0，两个传输缓冲区都没有被USB核心使用。

对于一个OUT端点，如果端点的最大尺寸小于该变量所指定的值，到USB设备的传输将被分解为更小的数据块以便正确地传输数据。这种大数据量的传输以连续的USB帧的方式进行。在一个urb中提交一个大数据块然后让USB主控制器把它分割为更小的块，比以连续的次序发送更小的缓冲区的速度快得多。

unsigned char *setup_packet

指向控制urb的设置数据包的指针。它在传输缓冲区中的数据之前被传送。该变量只对控制urb有效。

dma_addr_t setup_dma

控制urb用于设置数据包的DMA缓冲区。它在普通传输缓冲区中的数据之前被传送。该变量只对控制urb有效。

usb_complete_t complete

指向一个结束处理例程的指针，当urb被完全传输或者发生错误时，USB核心将调用该函数。在该函数内，USB驱动程序可以检查urb，释放它，或者把它重新提交到另一个传输中去。（有关结束处理例程的详情请参阅“结束urb：结束回调处理例程”一节）。

usb_complete_t的类型定义为：

```
typedef void (*usb_complete_t)(struct urb *, struct pt_regs *);
```

void *context

指向一个可以被USB驱动程序设置的数据块。它可以在结束处理例程中当urb被返回到驱动程序时使用。有关该变量的详情请参阅随后的小节。

int actual_length

当urb结束之后，该变量被设置为urb所发送的数据（OUT urb）或者urb所接收的数据（IN urb）的实际长度。对于IN urb，必须使用该变量而不是transfer_buffer_length变量，因为所接收的数据可能小于整个缓冲区的尺寸。

int status

当urb结束之后，或者正在被USB核心处理时，该变量被设置为urb的当前状态。USB驱动程序可以安全地访问该变量的唯一时刻是在urb结束处理例程中（在“结束urb：结束回调处理例程”一节中描述）。该限制是为了防止当urb正在被USB核心处理时竞态的发生。对于等时urb，该变量的一个成功值（0）只表示urb是否已经被解开链接。如果要获取等时urb的详细状态，应该检查iso_frame_desc变量。

该变量的有效值包括:

0

urb 传输成功。

-ENOENT

urb 被 *usb_kill_urb* 调用终止。

-ECONNRESET

urb 被 *usb_unlink_urb* 调用解开链接，urb 的 transfer_flags 变量被设置为 URB_ASYNC_UNLINK。

-EINPROGRESS

urb 仍然在被 USB 主控制器处理。如果驱动程序中检查到该值，说明存在代码缺陷。

-EPROTO

urb 发生了下列错误之一:

- 在传输中发生了 bitstuff 错误。
- 硬件没有及时接收到响应数据包。

-EILSEQ

urb 传输中发生了 CRC 校验不匹配。

-EPIPE

端点被中止。如果涉及的端点不是控制端点，可以调用 *usb_clear_halt* 函数来清除该错误。

-ECOMM

传输时数据的接收速度比把它写到系统内存的速度快。该错误值仅发生在 IN urb 上。

-ENOSR

传输时从系统内存获取数据的速度不够快，跟不上所要求的 USB 数据速率。该错误值仅发生在 OUT urb 上。

-EOVERFLOW

urb 发生了“串扰 (babble)”错误。“串扰”错误发生在端点接收了超过端点指定最大数据包尺寸的数据时。

-EREMOTEIO

仅发生在 urb 的 transfer_flags 变量被设置 URB_SHORT_NOT_OK 标志时，表示 urb 没有接收到所要求的全部数据量。

-ENODEV

USB 设备已从系统移除。

-EXDEV

仅发生在等时 urb 上，表示传输仅部分完成。为了确定所传输的内容，驱动程序必须查看单个帧的状态。

-EINVAL

urb 发生了很糟糕的事情。USB 内核文档描述了该值的含义：

> 等时错乱，如果发生这种情况：退出系统然后回家

如果 urb 结构体中的某一个参数没有被正确地设置或者 *usb_submit_urb* 调用中的不正确函数参数把 urb 提交到了 USB 核心，也可能发生这个错误。

-ESHUTDOWN

USB 主控制器驱动程序发生了严重的错误；设备已经被禁止，或者从系统脱离，而 urb 在设备被移除之后提交。如果当 urb 被提交到设备时设备的配置被改变，也可能发生这个错误。

一般来说，错误值 -EPROTO、-EILSEQ 和 -EOVERFLOW 表示设备、设备的固件或者把设备连接到计算机的电缆发生了硬件故障。

int start_frame

设置或者返回初始的帧数量，用于等时传输。

int interval

urb 被轮询的时间间隔。仅对中断或者等时 urb 有效。该值的单位随着设备速度的不同而不同。对于低速和满速的设备，单位是帧，相当于毫秒。对于其他设备，单位是微帧（microframe），相当于毫秒的 1/8。对于等时或者中断 urb，在 urb 被发送到 USB 核心之前，USB 驱动程序必须设置该值。

int number_of_packets

仅对等时 urb 有效，指定该 urb 所处理的等时传输缓冲区的数量。对于等时 urb，在 urb 被发送到 USB 核心之前，USB 驱动程序必须设置该值。

int error_count

由 USB 核心设置，仅用于等时 urb 结束之后。它表示报告了任何一种类型错误的等时传输的数量。

struct usb_iso_packet_descriptor iso_frame_desc[0]

仅对等时 urb 有效。该变量是一个 struct usb_iso_packet_descriptor 结构体数组。该结构体允许单个 urb 一次定义许多等时传输。它还用于收集每个单独传输的传输状态。

struct usb_iso_packet_descriptor由下列字段组成:

unsigned int offset
该数据包的数据在传输缓冲区中的偏移量（第一个字节为0）。

unsigned int length
该数据包的传输缓冲区大小。

unsigned int actual_length
该等时数据包接收到传输缓冲区中的数据长度。

unsigned int status
该数据包的单个等时传输的状态。它可以把相同的返回值作为主struct urb结构体的状态变量。

创建和销毁urb

struct urb结构体不能在驱动程序中或者另一个结构体中静态地创建，因为这样会破坏USB核心对urb所使用的引用计数机制。它必须使用*usb_alloc_urb*函数来创建。该函数原型如下:

```
struct urb *usb_alloc_urb(int iso_packets, int mem_flags);
```

第一个参数，iso_packets，是该urb应该包含的等时数据包的数量。如果不打算创建等时urb，该值应该设置为0。第二个参数，mem_flags，和传递给用于从内核分配内存的*kmalloc*函数（这些标志的详情参见第八章的“标志参数”一节）的标志有相同的类型。如果该函数成功地为urb分配了足够的内存空间，指向该urb的指针将被返回给调用函数。如果返回值为NULL，说明USB核心内发生了错误，驱动程序需要进行适当的清理。

当一个urb被创建之后，在它可以被USB核心使用之前必须被正确地初始化。关于如何初始化不同类型的urb，参见随后的小节。

驱动程序必须调用*usb_free_urb*函数来告诉USB核心驱动程序已经使用完urb。该函数只有一个参数:

```
void usb_free_urb(struct urb *urb);
```

这个参数是指向所需释放的struct urb的指针。在该函数被调用之后，urb结构体就消失了，驱动程序不能再访问它。

中断 urb

*usb_fill_int_urb*是一个辅助函数，用来正确地初始化即将被发送到USB设备的中断端点的 urb:

```
void usb_fill_int_urb(struct urb *urb, struct usb_device *dev,
                      unsigned int pipe, void *transfer_buffer,
                      int buffer_length, usb_complete_t complete,
                      void *context, int interval);
```

该函数包含很多的参数:

`struct urb *urb`
指向需初始化的 urb 的指针。

`struct usb_device *dev`
该 urb 所发送的目标 USB 设备。

`unsigned int pipe`
该 urb 所发送的目标 USB 设备的特定端点。该值是使用前述 *usb_sndintpipe* 或 *usb_rcvintpipe* 函数来创建的。

`void *transfer_buffer`
用于保存外发数据或者接收数据的缓冲区的指针。注意它不能是一个静态的缓冲区，必须使用 *kmalloc* 调用来创建。

`int buffer_length`
`transfer_buffer` 指针所指向的缓冲区的大小。

`usb_complete_t complete`
指向当该 urb 结束之后调用的结束处理例程的指针。

`void *context`
指向一个小数据块，该块被添加到 urb 结构体中以便进行结束处理例程后面的查找。

`int interval`
该 urb 应该被调度的间隔。有关该值的正确单位，请参考前面对 `struct urb` 结构体的描述。

批量 urb

批量urb的初始化和中断urb很相似。所使用的相关函数是*usb_fill_bulk_urb*，原型如下:

```
void usb_fill_bulk_urb(struct urb *urb, struct usb_device *dev,
                       unsigned int pipe, void *transfer_buffer,
```

```
                    int buffer_length, usb_complete_t complete,
                    void *context);
```

该函数的参数和 *usb_fill_int_urb* 函数完全一样。不过，没有时间间隔参数，因为批量urb没有时间间隔值。请注意，无符号整型pipe变量必须使用 *usb_sndbulkpipe* 或 *usb_rcvbulkpipe* 函数来初始化。

usb_fill_int_urb 函数不在urb中设置transfer_flags变量，因此，驱动程序必须自己来修改该字段。

控制urb

控制urb的初始化方法和批量urb几乎一样，调用 *usb_fill_control_urb* 函数。

```
void usb_fill_control_urb(struct urb *urb, struct usb_device *dev,
                          unsigned int pipe, unsigned char *setup_packet,
                          void *transfer_buffer, int buffer_length,
                          usb_complete_t complete, void *context);
```

函数参数和 *usb_fill_bulk_urb* 函数完全一样，除了一个新的参数，即unsigned char *setup_packet，它指向即将被发送到端点的设置数据包的数据。同样，无符号整型pipe变量必须使用 *usb_sndctrlpipe* 或 *usb_rcvictrlpipe* 函数来初始化。

usb_fill_control_urb 函数不设置urb中的transfer_flags变量，因此驱动程序必须自己修改该字段。大部分的驱动程序不使用该函数，因为使用“不使用urb的USB传输”一节中描述的同步API调用更加简单。

等时urb

不幸的是，等时urb没有和中断、控制和批量urb类似的初始化函数。因此它们在被提交到USB核心之前，必须在驱动程序中“手工地”进行初始化。下面是一个关于如何正确地初始化该类型urb的例子。它是从主内核源代码树的 *drivers/usb/media* 目录下的 *konicawc.c* 内核驱动程序中拿出来的。

```
urb->dev = dev;
urb->context = uvd;
urb->pipe = usb_rcvisocpipe(dev, uvd->video_endp-1);
urb->interval = 1;
urb->transfer_flags = URB_ISO_ASAP;
urb->transfer_buffer = cam->sts_buf[i];
urb->complete = konicawc_isoc_irq;
urb->number_of_packets = FRAMES_PER_DESC;
urb->transfer_buffer_length = FRAMES_PER_DESC;
for (j=0; j < FRAMES_PER_DESC; j++) {
        urb->iso_frame_desc[j].offset = j;
```

```
            urb->iso_frame_desc[j].length = 1;
    }
```

提交 urb

一旦 urb 被 USB 驱动程序正确地创建和初始化之后，就可以提交到 USB 核心以发送到 USB 设备了。这是通过调用 *usb_submit_urb* 函数来完成的。

```
int usb_submit_urb(struct urb *urb, int mem_flags);
```

urb参数是指向即将被发送到设备的urb的指针。mem_flags参数等同于传递给*kmalloc* 调用的同一个参数，用于告诉 USB 核心如何在此时及时地分配内存缓冲区。

当一个 urb 被成功地提交到 USB 核心之后，在接收函数被调用之前不能访问该 urb 结构体中的任何字段。因为 *usb_submit_urb* 函数可以在任何时刻调用（包括从一个中断上下文中），mem_flags 变量的内容必须是正确的。其实只有三个有效的值可以被使用，取决于 *usb_submit_urb* 何时被调用：

GFP_ATOMIC
: 只要下列条件成立就应该使用该值：

 - 调用者是在一个 urb 结束处理例程、中断处理例程、底半部、tasklet 或者定时器回调函数中。
 - 调用者正持有一个自旋锁或读写锁。注意如果持有了信号量，该值就不需要了。
 - current->state 不是 TASK_RUNNING。该状态永远是 TASK_RUNNING，除非驱动程序自己改变了当前的状态。

GFP_NOIO
: 如果驱动程序处于块 I/O 路径中应该使用该值。在所有存储类型的设备的错误处理路径中也应该使用它。

GFP_KERNEL
: 该值应该在前述类别之外的所有情况中使用。

结束 urb：结束回调处理例程

如果调用 *usb_submit_urb* 成功，把对 urb 的控制转交给 USB 核心，该函数返回 0；否则，返回负的错误号。如果函数调用成功，当 urb 结束的时候 urb 的结束处理例程（由结束函数指针指定）正好被调用一次。当该函数被调用时，USB 核心结束了对 URB 的处理，此刻对它的控制被返回给设备驱动程序。

只有三种结束urb和调用结束函数的情形:

- urb被成功地发送到了设备，设备返回了正确的确认。对于OUT urb而言就是数据被成功地发送，对于INT urb而言就是所请求的数据被成功地接收到。如果确实这样，urb中的status变量被设置为0。
- 发送数据到设备或者从设备接收数据时发生了某种错误。错误情况由urb结构体中的status变量的错误值来指示。
- urb从USB核心中被“解开链接”。当驱动程序通过*usb_unlink_urb*或*usb_kill_urb*调用告诉USB核心取消一个已提交的urb时，或者当设备从系统中被移除而一个urb已经提交给它时，会发生这种情况。

本章的稍后将给出一个如何在urb结束调用内检测各种不同的返回值的示例。

取消urb

应该调用*usb_kill_urb*或*usb_unlink_urb*函数来终止一个已经被提交到USB核心的urb。

```
int usb_kill_urb(struct urb *urb);
int usb_unlink_urb(struct urb *urb);
```

这两个函数的urb参数是指向即将被取消的urb的指针。

如果调用*usb_kill_urb*函数，该urb的生命周期将被终止。通常是当设备从系统中被断开时，在断开回调函数中调用该函数。

对于某些驱动程序而言，应该使用*usb_unlink_urb*函数来告诉USB核心终止一个urb。该函数并不等到urb完全被终止之后才返回到调用函数。这对于在中断处理例程中或者持有一个自旋锁时终止一个urb是很有用的，因为等待一个urb完全被终止需要USB核心具有使调用进程睡眠的能力。该函数需要被要求终止的urb中的URB_ASYNC_UNLINK标志值被设置才能正确地工作。

编写USB驱动程序

编写一个USB设备驱动程序的方法和pci_driver类似：驱动程序把驱动程序对象注册到USB子系统中，稍后再使用制造商和设备标识来判断是否已经安装了硬件。

驱动程序支持哪些设备?

struct usb_device_id结构体提供了一列不同类型的该驱动程序支持的USB设备。

USB核心使用该列表来判断对于一个设备该使用哪一个驱动程序，热插拔脚本使用它来确定当一个特定的设备插入到系统时该自动装载哪一个驱动程序。

struct usb_device_id结构体包括下列字段：

__u16 match_flags
: 确定设备和结构体中下列字段中的哪一个相匹配。这是一个由 *include/linux/mod_devicetable.h* 文件中指定的不同的USB_DEVICE_ID_MATCH_*值定义的位字段。通常不直接设置该字段，而是使用稍后介绍的USB_DEVICE类型的宏来初始化。

__u16 idVendor
: 设备的USB制造商ID。该编号是由USB论坛指派给其成员的，不会由其他人指定。

__u16 idProduct
: 设备的USB产品ID。所有指派了制造商ID的制造商都可以随意地赋予其产品ID。

__u16 bcdDevice_lo
__u16 bcdDevice_hi
: 定义了制造商指派的产品的版本号范围的最低和最高值。bcdDevice_hi值包括在内；该值是最高编号的设备的编号。这两个值都以二进制编码的十进制（BCD）格式来表示。这些变量，加上idVendor和idProduct，用来定义设备的特定版本号。

__u8 bDeviceClass
__u8 bDeviceSubClass
__u8 bDeviceProtocol
: 分别定义设备的类型、子类型和协议。这些编号由USB论坛指派，定义在USB规范中。这些值详细说明了整个设备的行为，包括该设备上的所有接口。

__u8 bInterfaceClass
__u8 bInterfaceSubClass
__u8 bInterfaceProtocol
: 和上述设备特定的值很类似，这些值分别定义类型、子类型和单个接口的协议。这些编号由USB论坛指派，定义在USB规范中。

kernel_ulong_t driver_info
: 该值不是用来比较是否匹配的，不过它包含了驱动程序在USB驱动程序的探测回调函数中可以用来区分不同设备的信息。

对于PCI设备，有许多用来初始化该结构体的宏：

USB_DEVICE(vendor, product)

创建一个struct usb_device_id结构体，仅和指定的制造商和产品ID值相匹配。该宏常用于需要一个特定驱动程序的USB设备。

USB_DEVICE_VER(vendor, product, lo, hi)

创建一个struct usb_device_id结构体，仅和某版本范围内的指定制造商和产品ID值相匹配。

USB_DEVICE_INFO(class, subclass, protocol)

创建一个struct usb_device_id结构体，仅和USB设备的指定类型相匹配。

USB_INTERFACE_INFO(class, subclass, protocol)

创建一个struct usb_device_id结构体，仅和USB接口的指定类型相匹配。

因此，对于一个只控制来自单一制造商的单一USB设备的简单USB设备驱动程序来说，struct usb_device_id表将被定义为：

```
/* 该驱动程序支持的设备列表 */
static struct usb_device_id skel_table [  ] = {
    { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },
    { }                     /* 终止入口项 */
};
MODULE_DEVICE_TABLE (usb, skel_table);
```

对于PC驱动程序，MODULE_DEVICE_TABLE宏是必需的，以允许用户空间的工具判断出该驱动程序可以控制什么设备。但是对于USB驱动程序来说，字符串usb必须是该宏中的第一个值。

注册USB驱动程序

所有USB驱动程序都必须创建的主要结构体是struct usb_driver。该结构体必须由USB驱动程序来填写，包括许多回调函数和变量，它们向USB核心代码描述了USB驱动程序。

struct module *owner

指向该驱动程序的模块所有者的指针。USB核心使用它来正确地对该USB驱动程序进行引用计数，使它不会在不合适的时刻被卸载掉。该变量应该被设置为THIS_MODULE宏。

const char *name

指向驱动程序名字的指针。在内核的所有USB驱动程序中它必须是唯一的，通常被设置为和驱动程序模块名相同的名字。如果该驱动程序运行在内核中，可以在sysfs的*/sys/bus/usb/drivers/*下面找到它。

const struct usb_device_id *id_table

指向struct usb_device_id表的指针，该表包含了一列该驱动程序可以支持的所有不同类型的USB设备。如果没有设置该变量，USB驱动程序中的探测回调函数不会被调用。如果想要驱动程序对于系统中的每一个USB设备都被调用，创建一个只设置driver_info字段的条目：

```
static struct usb_device_id usb_ids[  ] = {
    {.driver_info = 42},
    {  }
};
```

int (*probe) (struct usb_interface *intf, const struct usb_device_id *id)

指向USB驱动程序中的探测函数的指针。当USB核心认为它有一个struct usb_interface可以由该驱动程序处理时，它将调用该函数（在“探测和断开的细节”一节中描述）。USB核心用来作判断的指向struct usb_device_id的指针也被传递给该函数。如果USB驱动程序确认传递给它的struct usb_interface，它应该恰当地初始化设备然后返回0。如果驱动程序不确认该设备，或者发生了错误，它应该返回一个负的错误值。

void (*disconnect) (struct usb_interface *intf)

指向USB驱动程序中的断开函数的指针。当struct usb_interface被从系统中移除或者驱动程序正在从USB核心中卸载时，USB核心将调用该函数（在“探测和断开的细节”一节中描述）。

因此，创建一个有效的struct usb_driver结构体只需要初始化五个字段：

```
static struct usb_driver skel_driver = {
    .owner = THIS_MODULE,
    .name = "skeleton",
    .id_table = skel_table,
    .probe = skel_probe,
    .disconnect = skel_disconnect,
};
```

struct usb_driver还包含了另外几个回调函数，这些函数不是很常用，对于一个USB驱动程序的正常工作不是必需的：

int (*ioctl) (struct usb_interface *intf, unsigned int code, void *buf)

指向USB驱动程序中的*ioctl*函数。如果该函数存在，当用户空间的程序对*usbfs*文件系统中的设备文件进行了*ioctl*调用，而和该设备文件相关联的USB设备附着在该USB驱动程序上时，它将被调用。实际上，只有USB集线器驱动程序使用该ioctl，其他的USB驱动程序都没有使用它的真实需要。

int (*suspend) (struct usb_interface *intf, u32 state)

指向USB驱动程序中的挂起函数的指针。当设备将被USB核心挂起时调用该函数。

int (*resume) (struct usb_interface *intf)

指向USB驱动程序中的恢复函数的指针。当设备将被USB核心恢复时调用该函数。

以struct usb_driver指针为参数的*usb_register_driver*函数调用把struct usb_driver注册到USB核心。传统上是在USB驱动程序的模块初始化代码中完成该工作的:

```
static int __init usb_skel_init(void)
{
    int result;

    /* 把该驱动程序注册到USB子系统 */
    result = usb_register(&skel_driver);
    if (result)
        err("usb_register failed. Error number %d", result);

    return result;
}
```

当USB驱动程序将要被卸载时，需要把struct usb_driver从内核中注销。通过调用*usb_deregister_driver*来完成该工作。当该调用发生时，当前绑定到该驱动程序上的任何USB接口都被断开，断开函数将被调用。

```
static void __exit usb_skel_exit(void)
{
    /* 把该驱动程序从USB子系统注销 */
    usb_deregister(&skel_driver);
}
```

探测和断开的细节

上一节描述的struct usb_driver结构体中，驱动程序指定了两个USB核心在适当时间调用的函数。当一个设备被安装而USB核心认为该驱动程序应该处理时，探测函数被调用；探测函数应该检查传递给它的设备信息，确定驱动程序是否真的适合该设备。当驱动程序因为某种原因不应控制设备时，断开函数被调用，它可以做一些清理的工作。

探测和断开回调函数都是在USB集线器内核线程的上下文中被调用的，因此在其中睡眠是合法的。然而，建议大部分的工作尽可能地在用户打开设备时完成，从而把USB探测的时间减到最少。因为USB核心在单一线程中处理USB设备的添加和删除，任何低速的设备驱动程序都可以减慢USB设备的探测时间，从而影响用户的使用。

在探测回调函数中，USB 驱动程序应该初始化任何可能用于控制USB 设备的局部结构体。它还应该把所需的任何设备相关信息保存到局部结构体中，因为在此时做该工作是比较容易的。例如，USB驱动程序通常需要探测设备的端点地址和缓冲区大小，因为需要它们才能和设备进行通信。这里是一些示例代码，它们探测批量类型的IN 和OUT端点，把相关的信息保存到一个局部设备结构体中：

```
/* 设置端点信息 */
/* 只使用第一个批量 IN 和批量 OUT 端点 */
iface_desc = interface->cur_altsetting;
for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
    endpoint = &iface_desc->endpoint[i].desc;

    if (!dev->bulk_in_endpointAddr &&
        (endpoint->bEndpointAddress & USB_DIR_IN) &&
        ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
                =  = USB_ENDPOINT_XFER_BULK)) {
        /* 发现一个批量 IN 类型的端点 */
        buffer_size = endpoint->wMaxPacketSize;
        dev->bulk_in_size = buffer_size;
        dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
        dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);
        if (!dev->bulk_in_buffer) {
            err("Could not allocate bulk_in_buffer");
            goto error;
        }
    }

    if (!dev->bulk_out_endpointAddr &&
        !(endpoint->bEndpointAddress & USB_DIR_IN) &&
        ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
                =  = USB_ENDPOINT_XFER_BULK)) {
        /* 发现一个批量 OUT 类型的端点 */
        dev->bulk_out_endpointAddr = endpoint->bEndpointAddress;
    }
}
if (!(dev->bulk_in_endpointAddr && dev->bulk_out_endpointAddr)) {
    err("Could not find both bulk-in and bulk-out endpoints");
    goto error;
}
```

该代码块首先循环访问该接口中存在的每一个端点，赋予该端点结构体的局部指针以使稍后的访问更加容易：

```
for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
    endpoint = &iface_desc->endpoint[i].desc;
```

然后，在我们有了一个端点，而还没有发现批量IN类型的端点时，查看该端点的方向是否为IN。这可以通过检查位掩码USB_DIR_IN是否包含在bEndpointAddress端点变量中来确定。如果是的话，我们测定该端点类型是否批量，这通过首先以

USB_ENDPOINT_XFERTYPE_MASK位掩码来取bmAttributes变量的值，然后检查它是否和USB_ENDPOINT_XFER_BULK值匹配来完成：

```
if (!dev->bulk_in_endpointAddr &&
    (endpoint->bEndpointAddress & USB_DIR_IN) &&
    ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
            =  = USB_ENDPOINT_XFER_BULK)) {
```

如果所有这些检测都通过了，驱动程序就知道它已经发现了正确的端点类型，可以把该端点的相关信息保存到一个局部结构体中，以便稍后使用它来和端点进行通信：

```
/* 发现一个批量IN类型的端点 */
buffer_size = endpoint->wMaxPacketSize;
dev->bulk_in_size = buffer_size;
dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);
if (!dev->bulk_in_buffer) {
    err("Could not allocate bulk_in_buffer");
    goto error;
}
```

因为USB驱动程序需要在设备生命周期的稍后时间获取和该struct usb_interface相关联的局部数据结构体，所以可以调用*usb_set_intfdata*函数：

```
/* 把数据指针保存到这个接口设备中 */
usb_set_intfdata(interface, dev);
```

该函数接受一个指向任意数据类型的指针，把它保存到struct usb_interface结构体中以方便后面的访问。应该调用*usb_get_intfdata*函数来获取数据：

```
struct usb_skel *dev;
struct usb_interface *interface;
int subminor;
int retval = 0;

subminor = iminor(inode);

interface = usb_find_interface(&skel_driver, subminor);
if (!interface) {
    err ("%s - error, can't find device for minor %d",
         __FUNCTION__, subminor);
    retval = -ENODEV;
    goto exit;
}

dev = usb_get_intfdata(interface);
if (!dev) {
    retval = -ENODEV;
    goto exit;
}
```

usb_get_intfdata 通常在USB驱动程序的打开函数和断开函数中被调用。正是归功于这两个函数，USB驱动程序不需要维护一个静态的指针数组来存储系统中所有当前设备的设备结构体。对设备信息的非直接引用使得任何USB驱动程序都可以支持数量不限的设备。

如果USB驱动程序没有和处理设备与用户交互（例如输入、tty、视频等等）的另一种类型的子系统相关联，驱动程序可以使用USB主设备号，以便在用户空间使用传统的字符驱动程序接口。如果要这么做，USB驱动程序必须在探测函数中调用*usb_register_dev*函数来把设备注册到USB核心。只要该函数被调用，就要确保设备和驱动程序都处于可以处理用户访问设备的要求的恰当状态。

```
/* 现在可以注册设备了，它已准备好了 */
retval = usb_register_dev(interface, &skel_class);
if (retval) {
    /* 某些情况造成我们不能注册该驱动程序 */
    err("Not able to get a minor for this device.");
    usb_set_intfdata(interface, NULL);
    goto error;
}
```

*usb_register_dev*函数需要一个指向struct usb_interface的指针和一个指向struct usb_class_driver结构的指针。这个struct usb_class_driver用于定义许多不同的参数，在注册一个次设备号时USB驱动程序需要USB核心知道这些参数。该结构体包括如下的变量：

`char *name`

sysfs用来描述设备的名字。前导路径名，如果存在的话，只用在devfs中，本书不涉及该内容。如果设备的编号需要出现在名字中，名字字符串应该包含字符%d。例如，为了创建devfs名字usb/foo1和sysfs类型名字foo1，名字字符串应该设置为usb/foo%d。

`struct file_operations *fops;`

指向struct file_operations的指针，驱动程序定义该结构体，用它来注册为字符设备。有关该结构体的更多情况请参阅第三章。

`mode_t mode;`

为该驱动程序创建的devfs文件的模式；在其他情况下没有使用。该变量的一个典型设置是S_IRUSR和S_IWUSR值的组合，只提供了设备文件属主的读和写访问权限。

`int minor_base;`

这是为该驱动程序指派的次设备号范围的开始值。和该驱动程序相关联的所有设备

都是以唯一的、以该值开始的递增的次设备号来创建的。任何时刻只能允许有16个设备和该驱动程序相关联，除非内核的CONFIG_USB_DYNAMIC_MINORS配置选项被打开。如果如此，该变量将被忽略，以先来先办的方式来分配设备的所有次设备号。建议打开了该选项的系统使用类似udev这样的程序来管理系统中的设备节点，因为一个静态的*/dev*树不会工作正常。

当一个USB设备被断开时，和该设备相关联的所有资源都应该被尽可能地清理掉。在此时，如果已经在探测函数中调用了*usb_register_dev*来为该USB设备分配一个次设备号的话，必须调用*usb_deregister_dev*函数来把次设备号交还USB核心。

在断开函数中，从接口获取之前调用*usb_set_intfdata*设置的任何数据也是很重要的。然后设置struct usb_interface结构体中的数据指针为NULL，以防止任何不适当的对该数据的进行的错误访问。

```
static void skel_disconnect(struct usb_interface *interface)
{
    struct usb_skel *dev;
    int minor = interface->minor;

    /* 防止skel_open()和skel_disconnect()竞争 */
    lock_kernel();

    dev = usb_get_intfdata(interface);
    usb_set_intfdata(interface, NULL);

    /* 返回次设备号 */
    usb_deregister_dev(interface, &skel_class);

    unlock_kernel();

    /* 减小使用计数 */
    kref_put(&dev->kref, skel_delete);

    info("USB Skeleton #%d now disconnected", minor);
}
```

注意上述代码片段中的*lock_kernel*调用。它获取了大内核锁，以使断开回调函数在试图获取一个正确的接口数据结构体指针时不会和打开调用遭遇竞态。因为打开函数是在大内核锁被获取的情况下被调用的，如果断开函数也获取了同一个锁，驱动程序中只有一个部分可以访问和设置接口数据指针。

就在USB设备的断开回调函数被调用之前，所有正在传输到设备的urb都被USB核心取消，因此驱动程序不必要对这些urb显式地调用*usb_kill_urb*。在USB设备已经被断开之后，如果驱动程序试图通过调用*usb_submit_urb*来提交一个urb给它，提交将会失败并返回错误值-EPIPE。

提交和控制 urb

当驱动程序有数据要发送到USB设备时（典型地发生在驱动程序的写函数中），必须分配一个 urb 来把数据传输给设备：

```
urb = usb_alloc_urb(0, GFP_KERNEL);
if (!urb) {
    retval = -ENOMEM;
    goto error;
}
```

在 urb 被成功地分配之后，还应该创建一个 DMA 缓冲区来以最高效的方式发送数据到设备，传递给驱动程序的数据应该复制到该缓冲区中：

```
buf = usb_buffer_alloc(dev->udev, count, GFP_KERNEL, &urb->transfer_dma);
if (!buf) {
    retval = -ENOMEM;
    goto error;
}
if (copy_from_user(buf, user_buffer, count)) {
    retval = -EFAULT;
    goto error;
}
```

一旦数据从用户空间正确地复制到了局部缓冲区中，urb必须在可以被提交给USB核心之前被正确地初始化：

```
/* 正确地初始化 urb */
usb_fill_bulk_urb(urb, dev->udev,
          usb_sndbulkpipe(dev->udev, dev->bulk_out_endpointAddr),
          buf, count, skel_write_bulk_callback, dev);
urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
```

现在urb被正确地分配了，数据被正确地复制了，urb被正确地初始化了，它就可以被提交给 USB 核心以传输到设备：

```
/* 把数据从批量端口发出 */
retval = usb_submit_urb(urb, GFP_KERNEL);
if (retval) {
    err("%s - failed submitting write urb, error %d", __FUNCTION__, retval);
    goto error;
}
```

在urb被成功地传输到USB设备之后（或者传输中发生了某些事情），urb 回调函数将被USB核心调用。在我们的例子中，我们初始化urb，使之指向*skel_write_bulk_callback*函数，它就是被调用的函数：

```
static void skel_write_bulk_callback(struct urb *urb, struct pt_regs *regs)
{
```

```
        /* sync/async 解链接故障不是错误 */
        if (urb->status &&
            !(urb->status == -ENOENT ||
              urb->status == -ECONNRESET ||
              urb->status == -ESHUTDOWN)) {
            dbg("%s - nonzero write bulk status received: %d",
                __FUNCTION__, urb->status);
        }

        /* 释放已分配的缓冲区 */
        usb_buffer_free(urb->dev, urb->transfer_buffer_length,
                urb->transfer_buffer, urb->transfer_dma);
    }
```

回调函数中做的第一件事情是检查urb的状态，以确定该urb是否已经成功地结束。错误值，-ENOENT、-ECONNRESET和-ESHUTDOWN不是真的传输错误，只是报告一次成功的传输的相关情况（请参考“struct urb”一节中描述的urb可能的错误的列表）。之后回调函数释放传输时分配给该urb的缓冲区。

当urb回调函数正在运行时另一个urb被提交到设备是很常见的。这对于发送流式数据到设备很有用。不要忘了urb回调函数是运行在中断上下文中的，因此它不应该进行任何内存分配、持有任何信号量或者做任何其他可能导致进程睡眠的事情。当在回调函数内提交一个urb时，如果它在提交过程中需要分配新的内存块的话，使用GFP_ATOMIC标志来告诉USB核心不要睡眠。

不使用urb的USB传输

有时候USB驱动程序只是要发送或者接收一些简单的USB数据，而不想把创建一个struct urb、初始化它、然后等待该urb接收函数运行这些麻烦事都走一遍。有两个提供了更简单接口的函数可以使用。

usb_bulk_msg

*usb_bulk_msg*创建一个USB批量urb，把它发送到指定的设备，然后在返回调用者之前等待它的结束。它定义为：

```
int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe,
                 void *data, int len, int *actual_length,
                 int timeout);
```

该函数的参数为：

struct usb_device *usb_dev

指向批量消息所发送的目标USB设备的指针。

unsigned int pipe

该批量消息所发送的目标USB 设备的特定端点。该值是调用 *usb_sndbulkpipe* 或 *usb_rcvbulkpipe* 来创建的。

void *data

如果是一个OUT端点，它是指向即将发送到设备的数据的指针。如果是一个IN端点，它是指向从设备读取的数据应该存放的位置的指针。

int len

data 参数所指缓冲区的大小。

int *actual_length

指向保存实际传输字节数的位置的指针，至于是传输到设备还是从设备接收取决于端点的方向。

int timeout

以jiffies为单位的应该等待的超时时间。如果该值为0，该函数将一直等待消息的结束。

如果函数调用成功，返回值为0；否则，返回一个负的错误值。该错误值和“struct urb”一节中描述的urb错误编号相匹配。如果成功，actual_length参数包含从该消息发送或者接收的字节数。

下面是一个使用该函数调用的例子：

```
/* 进行阻塞的批量读以从设备获取数据 */
retval = usb_bulk_msg(dev->udev,
            usb_rcvbulkpipe(dev->udev, dev->bulk_in_endpointAddr),
            dev->bulk_in_buffer,
            min(dev->bulk_in_size, count),
            &count, HZ*10);

/* 如果读成功，复制数据到用户空间 */
if (!retval) {
    if (copy_to_user(buffer, dev->bulk_in_buffer, count))
        retval = -EFAULT;
    else
        retval = count;
}
```

该例子说明了一个从IN端点的简单的批量读。如果读取成功，数据被复制到用户空间。通常在USB 驱动程序的读函数中完成这个工作。

不能在一个中断上下文中或者在持有自旋锁的情况下调用*usb_bulk_msg*函数。同样，该函数不能被任何其他函数取消，因此使用它的时候要小心；确保驱动程序的断开函数了解足够的信息，在允许自身从内存中被卸载之前等待该调用的结束。

usb_control_msg

除了允许驱动程序发送和接收USB控制消息之外，*usb_control_msg*函数的运作和*usb_bulk_msg*函数类似：

```
int usb_control_msg(struct usb_device *dev, unsigned int pipe,
                    __u8 request, __u8 requesttype,
                    __u16 value, __u16 index,
                    void *data, __u16 size, int timeout);
```

该函数的参数和*usb_bulk_msg*很相似，但有几个重要的区别：

`struct usb_device *dev`
指向控制消息所发送的目标USB设备的指针。

`unsigned int pipe`
该控制消息所发送的目标USB设备的特定端点。该值是调用usb_sndctrlpipe或usb_rcvctrlpipe来创建的。

`__u8 request`
控制消息的USB请求值。

`__u8 requesttype`
控制消息的USB请求类型值。

`__u16 value`
控制消息的USB消息值。

`__u16 index`
控制消息的USB消息索引值。

`void *data`
如果是一个OUT端点，它是指向即将发送到设备的数据的指针。如果是一个IN端点，它是指向从设备读取的数据应该存放的位置的指针。

`__u16 size`
data参数所指缓冲区的大小。

`int timeout`
以jiffies为单位的应该等待的超时时间。如果该值为0，该函数将一直等待消息的结束。

如果函数调用成功，它返回传输到设备或者从设备读取的字节数；如果不成功，它返回一个负的错误值。

request、requesttype、value 和 index 参数都直接映射到 USB 规范的 USB 控制消息定义处。关于这些参数的有效值和如何使用，请参考 USB 规范的第九章。

和 *usb_bulk_msg* 函数一样，*usb_control_msg* 函数不能在一个中断上下文中或者持有自旋锁的情况下调用。同样，该函数不能被任何其他函数取消，因此使用它的时候要小心；确保驱动程序的断开函数了解足够的信息，在允许自身从内存中被卸载之前等待该调用的结束。

其他 USB 数据函数

USB 核心中的许多辅助函数可以用来从所有 USB 设备中获取标准的信息。这些函数不能在一个中断上下文中或者持有自旋锁的情况下调用。

usb_get_descriptor 函数从指定的设备获取指定的 USB 描述符。该函数定义为：

```
int usb_get_descriptor(struct usb_device *dev, unsigned char type,
                       unsigned char index, void *buf, int size);
```

USB 驱动程序可以使用该函数来从 struct usb_device 结构体中获取任何没有存在于已有 struct usb_device 和 struct usb_interface 结构体中的设备描述符，例如音频描述符或者其他的类型特定信息。该函数的参数为：

struct usb_device *usb_dev
: 指向想要获取描述符的目标 USB 设备的指针。

unsigned char type
: 描述符的类型。该类型在 USB 规范中有描述，可以是下列类型中的一种：

```
USB_DT_DEVICE
USB_DT_CONFIG
USB_DT_STRING
USB_DT_INTERFACE
USB_DT_ENDPOINT
USB_DT_DEVICE_QUALIFIER
USB_DT_OTHER_SPEED_CONFIG
USB_DT_INTERFACE_POWER
USB_DT_OTG
USB_DT_DEBUG
USB_DT_INTERFACE_ASSOCIATION
USB_DT_CS_DEVICE
USB_DT_CS_CONFIG
USB_DT_CS_STRING
USB_DT_CS_INTERFACE
USB_DT_CS_ENDPOINT
```

unsigned char index
应该从设备获取的描述符的编号。

void *buf
指向复制描述符到其中的缓冲区的指针。

int size
buf 变量所指内存的大小。

如果该函数调用成功，它返回从设备读取的字节数。否则，它返回一个由该函数调用的底层的 *usb_control_msg* 函数返回的一个负的错误值。

usb_get_descriptor 调用更常用于从USB设备获取一个字符串。因为这很常见，所以提供了一个名为 *usb_get_string* 的辅助函数来完成该工作：

```
int usb_get_string(struct usb_device *dev, unsigned short langid,
                   unsigned char index, void *buf, int size);
```

如果成功，该函数返回从设备接收的字符串的字节数。否则，它返回一个由该函数调用的底层的 *usb_control_msg* 函数返回的一个负的错误值。

如果该函数调用成功，它返回一个以UTF-16LE格式（Unicode，每个字符16位，小端字节序）编码的字符串，保存在buf参数所指的缓冲区中。因为这种格式不是很有用，有另一个名为 *usb_string* 的函数返回从USB设备读取的已经转换为ISO 8859-1格式的字符串。这种字符集是Unicode的一个8位的子集，是英语和其他西欧语言字符串的最常见格式。因为它是USB设备的字符串的典型格式，建议使用 *usb_string* 函数而不是 *usb_get_string* 函数。

快速参考

本节总结本章中介绍的符号：

#include <linux/usb.h>
和USB相关的所有内容所在的头文件。所有的USB设备驱动程序都必须包括该文件。

struct usb_driver;
描述USB驱动程序的结构体。

struct usb_device_id;
描述该驱动程序支持的USB设备类型的结构体。

```
int usb_register(struct usb_driver *d);
void usb_deregister(struct usb_driver *d);
```

用于往 USB 核心注册和注销 USB 驱动程序的函数。

```
struct usb_device *interface_to_usbdev(struct usb_interface *intf);
```

从一个 struct usb_interface * 获取一个控制的 struct usb_device *。

```
struct usb_device;
```

控制整个 USB 设备的结构体。

```
struct usb_interface;
```

主要的 USB 设备结构体，所有的 USB 驱动程序都用它来和 USB 核心进行通信。

```
void usb_set_intfdata(struct usb_interface *intf, void *data);
void *usb_get_intfdata(struct usb_interface *intf);
```

用于设置和获取 struct usb_interface 内私有数据指针段的函数。

```
struct usb_class_driver;
```

描述了想要使用 USB 主设备号和用户空间程序进行通信的 USB 驱动程序的结构体。

```
int usb_register_dev(struct usb_interface *intf, struct usb_class_driver
                     *class_driver);
void usb_deregister_dev(struct usb_interface *intf, struct usb_class_driver
                        *class_driver);
```

用于注册和注销特定的 struct usb_interface *结构体的函数，使用一个 struct usb_class_driver * 结构体。

```
struct urb;
```

描述一个 USB 数据传输的结构体。

```
struct urb *usb_alloc_urb(int iso_packets, int mem_flags);
void usb_free_urb(struct urb *urb);
```

用于创建和销毁一个 struct urb * 的函数。

```
int usb_submit_urb(struct urb *urb, int mem_flags);
int usb_kill_urb(struct urb *urb);
int usb_unlink_urb(struct urb *urb);
```

用于开始和终止一个 USB 数据传输。

```
void usb_fill_int_urb(struct urb *urb, struct usb_device *dev, unsigned int
  pipe, void *transfer_buffer, int buffer_length, usb_complete_t complete,
  void *context, int interval);
void usb_fill_bulk_urb(struct urb *urb, struct usb_device *dev, unsigned int
  pipe, void *transfer_buffer, int buffer_length, usb_complete_t complete,
  void *context);
void usb_fill_control_urb(struct urb *urb, struct usb_device *dev, unsigned
  int pipe, unsigned char *setup_packet, void *transfer_buffer, int
  buffer_ length, usb_complete_t complete, void *context);
```

用于在一个 struct urb 被提交到USB 核心之前对它进行初始化的函数。

```
int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe, void *data,
  int len, int *actual_length, int timeout);
int usb_control_msg(struct usb_device *dev, unsigned int pipe, __u8 request,
  __u8 requesttype, __u16 value, __u16 index, void *data, __u16 size,
  int timeout);
```

用于在不使用 struct urb 的情况下发送或接收USB 数据的函数。



第十四章

Linux 设备模型

在内核2.5的开发周期中需要完成的一个目标是：为内核建立起一个统一的设备模型。在以前的内核中没有独立的数据结构用来让内核获得系统整体配合的信息。尽管缺乏这些信息，在许多时候还是能工作正常。然而，随着拓扑结构越来越复杂，以及要支持诸如电源管理等新特性的需求，向新版本的内核明确提出了这样的要求：需要有一个对系统结构的一般性抽象描述。

2.6版的设备模型提供了这样的抽象。现在内核使用该抽象支持了多种不同的任务，其中包括：

电源管理和系统关机

完成这些工作需要一些对系统结构的理解。比如一个USB宿主适配器，在处理完所有与其连接的设备前是不能被关闭的。设备模型使得操作系统能够以正确的顺序遍历系统硬件。

与用户空间通信

sysfs虚拟文件系统的实现与设备模型密切相关，并且向外界展示了它所表述的结构。向用户空间所提供的系统信息，以及改变操作参数的接口，将越来越多地通过sysfs实现，也就是说，通过设备模型实现。

热插拔设备

越来越多的计算机设备可被动态的热插拔了，也就是说，外围设备可根据用户的需要安装与卸载。内核中的热插拔机制可以处理热插拔设备，特别是能够与用户空间进行关于插拔设备的通信，而这种机制也是通过设备模型管理的。

设备类型

系统中的许多部分对设备如何连接的信息并不感兴趣，但是它们需要知道哪些类型的设备是可以使用的。设备模型包括了将设备分类的机制，它会在更高的功能层上描述这些设备，并使得这些设备对用户空间可见。

对象生命周期

上述许多功能，包括热插拔支持和sysfs，使得内核中创建和管理对象的工作更为复杂。设备模型的实现需要创建一系列机制以处理对象的生命周期、对象之间的关系，以及这些对象在用户空间中的表示。

Linux设备模型是一个复杂的数据结构。举个例子，请看图14-1。图中显示了与USB鼠标相关联的设备模型的一小部分（以简化的形式给出）。在图的中央，可以看到核心“设备”树的一部分，它表明了鼠标是如何连接到系统的。“总线”树跟踪了连接到每个总线的设备，在“类”下的子树更关心设备所提供的功能，而不是设备是如何连接的。即使是一个简单的系统设备模型也包含了几百个如这个图那样的节点，这样，把它们全部展现出来需要一个很复杂的数据结构。

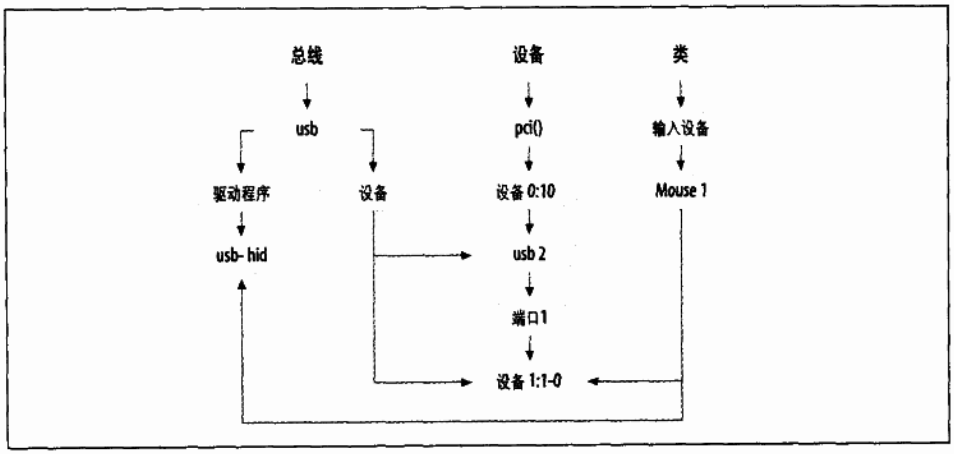


图 14-1：设备模型的一小片段

对模型的大部分来说，Linux设备模型代码会处理好这些关系，而不把它们强加给驱动程序的作者。模型隐藏在交互背后，与设备模型的直接交互通常由总线级逻辑和其他内核子系统来处理。其结果是，许多驱动程序的作者可完全忽略设备模型，并相信设备模型能处理好它所负责的事情。

了解设备模型是大有裨益的。设备模型有时会从其他层的背后暴露出来，比如，一般性的DMA代码（第十五章将讨论）使用了device结构。用户可能要使用设备模型所提供的一些功能，比如kobject所提供的引用计数及其相关功能。通过sysfs与用户空间通信也是设备模型的功能，本章将讨论这种通信的工作原理。

本章将对设备模型从下向上进行讲述。设备模型的复杂性使得从较高的层次来理解是相

对困难的。这里只是希望读者通过了解底层设备组件的工作原理，并在知识上做一些准备，使读者能理解这些组件是如何创建一个大型结构的。

对于许多读者来说，本章的内容可以认为是高级教材，在阅读第一遍的时候可以跳过本章。而对于那些十分想了解Linux设备模型是如何工作的读者，由于本章将深入研究其底层，因此十分值得好好学习。

kobject、kset 和子系统

*kobject*是组成设备模型的基本结构。最初它只是被理解为一个简单的引用计数，但是随着时间的推移，它的任务越来越多、因此也有了许多成员。现在kobject结构所能处理的任务以及它所支持的代码包括：

对象的引用计数
: 通常，一个内核对象被创建时，不可能知道该对象存活的时间。跟踪此对象生命周期的一个方法是使用引用计数。当内核中没有代码持有该对象的引用时，该对象将结束自己的有效生命周期，并且可以被删除。

sysfs 表述
: 在sysfs中显示的每一个对象，都对应一个kobject，它被用来与内核交互并创建它的可见表述。

数据结构关联
: 从整体上看，设备模型是一个友好而复杂的数据结构，通过在其间的大量连接而构成一个多层次的体系结构。kobject 实现了该结构并把它们聚合在一起。

热插事件拔处理
: 当系统中的硬件被热插拔时，在kobject子系统控制下，将产生事件以通知用户空间。

从前面的介绍中，读者可能会得出kobject是一个复杂结构的结论。的确是这样的。每次只去理解该结构的一部分，这样对了解该结构是如何工作的更为可行。

kobject 基础知识

kobject是一种数据结构，它定义在<linux/kobject.h>中。在这个文件中，还包括了与kobject相关结构的声明，当然还有一个用于操作kobject对象的函数清单。

嵌入的 kobject

在深入细节研究前，花点时间了解kobject的工作过程是值得的。如果回头看看kobject所处理的函数清单，就能发现它们都是一些代表其他对象完成的服务。换句话说，一个kobject对自身并不感兴趣，它存在的意义在于把高级对象连接到设备模型上。

因此，内核代码很少（甚至不知道）去创建一个单独的kobject对象，相反，kobject用于控制对大型域（domain）相关对象的访问。为了达到此目的，我们会发现kobject对象被嵌入到其他结构中。如果读者熟悉使用面向对象的方法思考，kobject可以被认为是最顶层的基类，其他类都是它的派生产物。kobject实现了一系列的方法，对自身并没有特殊的作用，但是对其他对象却非常有效。在C语言中不允许直接描述继承关系，因此使用了诸如在一个结构中嵌入另外一个结构的技术。

举一个例子，先回头看看在第三章遇到过的cdev结构。该结构在2.6.10内核中有着如下形式：

```
struct cdev {
    struct kobject kobj;
    struct module *owner;
    struct file_operations *ops;
    struct list_head list;
    dev_t dev;
    unsigned int count;
};
```

如上所见，cdev结构中嵌入了kobject结构。如果使用该结构，只需要访问kobject成员就能获得嵌入的kojbect对象。使用kobject的代码经常遇到相反的问题：对于给定的一个kobject指针，如何获得包含它的结构指针呢？必须要抛弃一些想当然的想法（比如假设kobject处于包含结构开始的位置），此时要使用*container_of*宏（在第三章“open函数”一节中有过讲述）。利用这个宏，对包含在cdev结构中的、名为kp的kobject结构指针进行转换的代码如下：

```
struct cdev *device = container_of(kp, struct cdev, kobj);
```

为了能通过kobject指针回找（back-casting）包含它的类，程序员经常要定义简单的宏完成这件事。

kobject的初始化

为了在编译和运行时对结构进行初始化，本书提供了大量的简单机制。但是对kobject的初始化要复杂一些，特别是当kobject所有的函数都要被用到时，其初始化更加复杂。不管如何使用kobject，有一些步骤是必须的。

首先是将整个kobject设置为0，这通常使用*memset*函数。通常在对包含kobject的结构清零时，使用这种初始化方法。如果忘记对kobject的清零初始化，则在以后使用kobject时，可能会发生一些奇怪的错误，因此，不能跳过这一步骤。

之后调用*kobject_init()*函数，以便设置结构内部的一些成员：

```
void kobject_init(struct kobject *kobj);
```

*kobject_init*所做的一件事情是设置kobject的引用计数为1。然而仅仅调用*kobject_init*是不够的。kobject的使用者必须至少设置kobject的名字，这是在sysfs入口中使用的名字。如果仔细分析内核源代码，可以发现直接将字符串拷贝到kobject的name成员的代码。但尽量别这么做，而应该使用：

```
int kobject_set_name(struct kobject *kobj, const char *format, ...);
```

该函数使用了类似*printk*的变量参数列表。不管是否相信，它可能会导致该操作的失败（因为要分配内存），因此，严格的代码应该检查返回值，并做相应的处理。

kobject的创建者需要直接或者间接设置的成员有：ktype、kset和parent。在本章以后的部分中，将对它们进行讲解。

对引用计数的操作

kobject的一个重要函数是为包含它的结构设置引用计数。只要对象的引用计数存在，对象（以及支持它的代码）就必须继续存在。底层控制kobject引用计数的函数有：

```
struct kobject *kobject_get(struct kobject *kobj);
void kobject_put(struct kobject *kobj);
```

对*kobject_get*的成功调用将增加kobject的引用计数，并返回指向kobject的指针。如果kobject已经处于被销毁的过程中，则该调用失败，*kobject_get*返回NULL。必须检查返回值，否则可能会产生麻烦的竞态。

当引用被释放时，调用*kobject_put*减少引用计数，并在可能的情况下释放该对象。请记住*kobject_init*设置引用计数为1，所以当创建kobject时，如果不再需要初始的引用，就要调用相应的*kobject_put*函数。

请注意，在许多情况下，在kobject中的引用计数不足以防止竞态的产生。举例来说，kobject（以及包含它的结构）的存在需要创建它的模块继续存在。当kobject继续被使用时，不能卸载该模块。这就是为什么在前面看到的cdev结构中包含了模块指针的原因。cdev结构中引用计数的实现代码如下：

```
struct kobject *cdev_get(struct cdev *p)
{
    struct module *owner = p->owner;
    struct kobject *kobj;

    if (owner && !try_module_get(owner))
        return NULL;
    kobj = kobject_get(&p->kobj);
    if (!kobj)
        module_put(owner);
    return kobj;
}
```

创建对 cdev 结构的引用时，也需要创建包含它的模块的引用。因此，*cdev_get* 使用 *try_module_get* 去增加模块的使用计数。如果操作成功，使用 *kobject_get* 增加 kobject 的引用计数。当然这个操作也可能会失败，因此代码需要检查 *kobject_get* 的返回值。如果调用失败，则要释放对模块的引用计数。

release 函数和 kobject 类型

在上面的讨论中还漏掉了一个重要内容，就是当引用计数为0的时候，kobject将采取什么操作。通常，创建 kobject 的代码无法知道这种情况会在什么时候发生。如果能知道的话，使用引用计数就毫无意义。在使用 sysfs 的时候，即使那些可预知的对象生命期也会变得更为复杂，因为用户空间程序可能在任意时间内引用 kobject 对象（比如让对应的 sysfs 文件保持打开状态）。

最终结果是，一个被 kobject 所保护的结构，不能在驱动程序生命周期的任何可预知的、单独的时间点上被释放掉。但是当 kobject 的引用计数为0时，上述代码又要随时准备运行。引用计数不为创建 kobject 的代码所直接控制。因此当 kobject 的最后一个引用计数不再存在时，必须异步地通知。

通知是使用 kobject 的 *release* 方法实现的，该方法通常的原型如下：

```
void my_object_release(struct kobject *kobj)
{
    struct my_object *mine = container_of(kobj, struct my_object, kobj);
    /* 对该对象执行其他的清除工作，然后…… */
    kfree(mine);
}
```

有一个要点不能被忽略：每个 kobject 都必须有一个 *release* 方法，并且 kobject 在该方法被调用前必须保持不变（处于稳定状态）。如果不能满足这些限制，代码中就会存在缺陷。当对象还在被使用的时候就释放它，则非常危险；或者，在最后一个引用返回前释放对象，该操作将失败。

有意思的是，*release* 函数并没有包含在kobject自身内，相反，它是与包含kobject的结构类型相关联的。一种称为ktype的kobj_type数据结构负责对该类型进行跟踪。下面是kobj_type结构的声明：

```
struct kobj_type {
    void (*release)(struct kobject *);
    struct sysfs_ops *sysfs_ops;
    struct attribute **default_attrs;
};
```

在kobj_type的release成员中，保存的是这种kobject类型的*release*函数指针。在本章中还要讲解另外两个成员（sysfs_ops和default_attrs）。

每个kobject都需要有一个相应的kobj_type结构。另人困惑的是，可以在两个不同的地方找到这个结构的指针。在kobject结构中包含了一个成员（称之为ktype）保存了该指针。但是，如果kobject是kset的一个成员的话，kset会提供kobj_type指针（在下一节中会讲到kset）。如下的宏：

```
struct kobj_type *get_ktype(struct kobject *kobj);
```

查找指定kobject的kobj_type指针。

kobject层次结构、kset和子系统

通常，内核用kojbect结构将各个对象连接起来组成一个分层的结构体系，从而与模型化的子系统相匹配。有两种独立的机制用于连接：parent指针和kset。

在kobject结构的parent成员中，保存了另外一个kobject结构的指针，这个结构表示了分层结构中上一层的节点。比如一个kobject结构表示了一个USB设备，它的parent指针可能指向了表示USB集线器的对象，而USB设备是插在USB集线器上的。

对parent指针最重要的用途是在sysfs分层结构中定位对象。在后面的“低层sysfs操作”一节中，读者将会看到它是如何实现的。

kset

从许多角度上看，kset像是kobj_type结构的扩充。一个kset是嵌入相同类型结构的kobject集合。但kobj_type结构关心的是对象的类型，而kset结构关心的是对象的聚集与集合。这两个概念是分立的。这样同种类型的对象可以出现在不同的集合中。

因此kset的主要功能是包容；我们可以认为它是kobject的顶层容器类。实际上，在每个kset内部，包含了自己的kobject，并且可以用多种处理kobject的方法处理kset。需

要注意的是，kset总是在sysfs中出现；一旦设置了kset并把它添加到系统中，将在sysfs中创建一个目录。kobject不必在sysfs中表示，但是kset中的每一个kobject成员都将在sysfs中得到表述。

创建一个对象时，通常要把一个kobject添加到kset中去。这个过程有两个步骤。先把kobject的kset成员要指向目的kset，然后将kobject传递给下面的函数：

```
int kobject_add(struct kobject *kobj);
```

和处理相似的函数一样，程序员应该意识到该函数可能会失败（如果失败，将返回一个负的错误码），并对此做出相应的动作。内核提供了一个方便使用的函数：

```
extern int kobject_register(struct kobject *kobj);
```

该函数只是*kobject_init*和*kobject_add*的简单组合。

当把一个kobject传递给*kobject_add*时，将会增加它的引用计数。在kset中包含的最重要的内容是对象的引用。在某些时候，可能不得不把kobject从kset中删除，以清除引用；使用下面的函数达到这个目的：

```
void kobject_del(struct kobject *kobj);
```

还有一个*kobject_unregister*函数，它是*kobject_del*和*kobject_put*的组合。

kset在一个标准的内核链表中保存了它的子节点。在大多数情况下，所包含的kobject会在它们的parent成员中保存kset（严格地说是其内嵌的kobject）的指针。因此典型的情况是，kset和它的kobject的关系与图14-2所示类似。请记住：

- 在图中所有被包含的kobject，实际上是被嵌入到其他类型中，甚至可能是其他的kset。
- 一个kobject的父节点不一定是包含它的kset（这样的结构非常少见）。

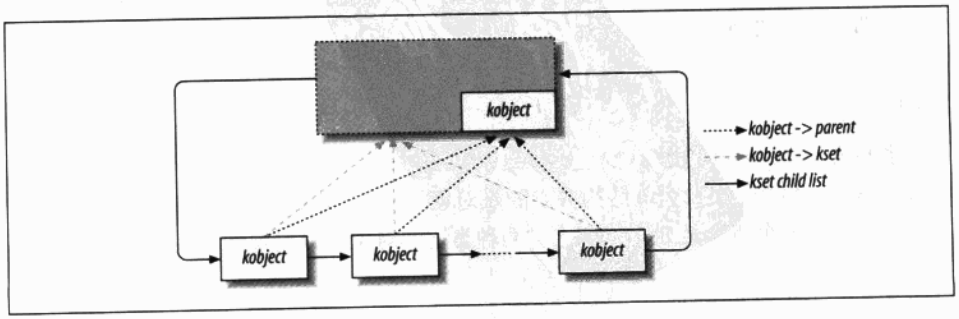


图14-2：一个简单的kset分层结构

kset 上的操作

kset 拥有与 kobject 相似的初始化和设置接口。下面是这些函数:

```
void kset_init(struct kset *kset);
int kset_add(struct kset *kset);
int kset_register(struct kset *kset);
void kset_unregister(struct kset *kset);
```

在大多数情况下，这些函数只是对kset中的kobject结构调用类似前面 *kobject_* 的函数。

为了管理 kset 的引用计数，其情况也是一样的:

```
struct kset *kset_get(struct kset *kset);
void kset_put(struct kset *kset);
```

一个kset也拥有名字，它保存在内嵌的kobject中。因此，如果我们有一个名为my_set的kset，可使用下面的函数设置它的名字:

```
kobject_set_name(&my_set->kobj, "The name");
```

kset中也有一个指针（在ktype成员中）指向kobj_type结构，用来描述它所包含的kojbect。该类型的使用优先于kobject中的ktype。因此在典型应用中，kobject中的ktype成员被设置为NULL。因为kset中的ktype成员是实际上被使用的成员。

最后，kset包含了一个子系统指针（称之为subsys）。因此现在该讲一讲子系统了。

子系统

子系统是对整个内核中一些高级部分的表述。子系统通常（但不一定）显示在sysfs分层结构中的顶层。内核中的子系统包括block_subsys（对块设备来说是*/sys/block*）、devices_subsys（*/sys/devices*，设备分层结构的核心）以及内核所知晓的用于各种总线的特定子系统。一个驱动程序的作者几乎不需要创建一个新的子系统。如果想这么做的话，请三思而后行。驱动程序作者最终要做的是添加一个新类，这如同在“类”一节中所讲的那样。

下面的简单结构表示了一个子系统:

```
struct subsystem {
    struct kset kset;
    struct rw_semaphore rwsem;
};
```

一个子系统其实是对kset和一个信号量的封装。

每一个 kset 都必须属于一个子系统。子系统的成员将帮助内核在分层结构中定位 kset，但更重要的是，子系统的 rwsem 信号量被用于串行访问 kset 内部的链表。在 kset 结构中，这种成员关系被表示为 subsys 指针。因此通过 kset 结构，可以找到包含 kset 的每一个子系统。但是我们无法直接从 subsystem 结构中找到子系统所包含的多个 kset。

通常使用下面的宏声明 subsystem:

```
decl_subsys(name, struct kobj_type *type,
            struct kset_hotplug_ops *hotplug_ops);
```

该宏使用传递给它的name并追加_subsys，然后做为结构名而创建sybsystem结构。该宏还使用了指定的 type 和 hotplug_ops 初始化内部的 kset（在本章后面的部分将讲到热插拔操作）。

子系统拥有一个设置和销毁的函数列表:

```
void subsystem_init(struct subsystem *subsys);
int subsystem_register(struct subsystem *subsys);
void subsystem_unregister(struct subsystem *subsys);
struct subsystem *subsys_get(struct subsystem *subsys)
void subsys_put(struct subsystem *subsys);
```

这些函数中的大多数都用于对子系统中的 kset 进行操作。

低层 sysfs 操作

kobject 是隐藏在 sysfs 虚拟文件系统后的机制，对于 sysfs 中的每个目录，内核中都会存在一个对应的 kobject。每一个 kobject 都输出一个或者多个属性，它们在 kobject 的 sysfs 目录中表现为文件，其中的内容由内核生成。这部分内容揭示了 kobject 和 sysfs 在底层是如何交互的。

<linux/sysfs.h> 中包含了 sysfs 的工作代码。

只要调用 *kobject_add*，就能在 sysfs 中显示 kobject。在把 kobject 添加到 kset 的时候，已经讨论过这个函数了；在 sysfs 中创建入口项也是该函数的功能之一。需要了解许多知识，才能理解是如何创建 sysfs 入口的。

- kobject 在 sysfs 中的入口始终是一个目录，因此，对 *kobject_add* 的调用将在 sysfs 中创建一个目录。通常这个目录包含一个或多个属性，随后将讲到如何设置属性。
- 分配给 kobject（使用 *kobject_set_name* 函数）的名字是 sysfs 中的目录名。这样，处于 sysfs 分层结构相同部分中的 kobject 必须有唯一的名字。分配给 kobject 的名字必须是合法的文件名：不能包含反斜杠，并且强烈建议不要使用空格。

- sysfs 入口在目录中的位置对应于 kobject 的 parent 指针。如果调用 *kobject_add* 的时候，parent 是 NULL，它将被设置为嵌入到新 kobject 的 kset 中的 kobject，这样，sysfs 分层结构通常与 kset 创建的内部结构相匹配。如果 parent 和 kset 都是 NULL，则会在最高层创建 sysfs 目录，而这通常不是我们所期望的。

使用前面介绍过的机制，可以使用 kobject 在 sysfs 中创建空目录。有时读者可能想做一些更有趣的事，因此现在需要讲述一下属性的实现方法。

默认属性

当创建 kobject 的时候，都会给每个 kobject 一系列默认属性。这些属性保存在 kobj_type 结构中。下面是该结构的成员：

```
struct kobj_type {
    void (*release)(struct kobject *);
    struct sysfs_ops *sysfs_ops;
    struct attribute **default_attrs;
};
```

default_attrs 成员保存了属性列表，用于创建该类型的每一个 kobject，sysfs_ops 提供了实现这些属性的方法。先来研究 default_attrs，它指向了一个包含 attribute 结构数组的指针：

```
struct attribute {
    char *name;
    struct module *owner;
    mode_t mode;
};
```

在这个结构中，name 是属性的名字（在 kobject 的 sysfs 目录中显示），owner 是指向模块的指针（如果有的话），该模块负责实现这些属性，mode 是应用于属性的保护位。对于只读属性，mode 通常是 S_IRUGO。如果属性是可写的，则可以使用 S_IWUSR 仅为 root 提供写权限（操作模式的宏在 *<linux/stat.h>* 中定义）。default_attrs 链表中的最后一个元素必须用零填充。

default_attrs 数组说明了都有些什么属性，但是没有告诉 sysfs 如何真正实现这些属性。这个任务交给了 kobj_type->sysfs_ops 成员，它所指向的结构定义如下：

```
struct sysfs_ops {
    ssize_t (*show)(struct kobject *kobj, struct attribute *attr,
                    char *buffer);
    ssize_t (*store)(struct kobject *kobj, struct attribute *attr,
                     const char *buffer, size_t size);
};
```

当用户空间读取一个属性时，内核会使用指向 kobject 的指针和正确的属性结构来调用 *show* 方法。该方法将把指定的值编码后放入缓冲区，然后把实际长度做为返回值返回。请注意不要越界（它有 PAGE_SIZE 个字节大）。sysfs 的约定要求每个属性都要包含一个单个的人眼可阅读的值。如果要返回大量的信息，则需要把它拆分成多个属性。

也可以对所有的 kobject 的属性使用同一个 *show* 方法。传递给该函数的 attr 指针可以用来判断所请求的是哪个属性。一些 *show* 方法包含一系列对属性名的检查。其他的实现方法会把 attribute 结构嵌入到其他结构中，而在那些结构中包含了需要返回的属性值信息，在这种情况下，可在 *show* 方法中使用 *container_of*，以获得嵌入结构的指针。

store 函数与此类似；它将对保存在缓冲区中的数据解码（在 size 中保存了数据的长度，该长度不能超过 PAGE_SIZE），并调用各种实用的方法保存新值，并且返回实际解码的字节数。只有当拥有属性的写权限时，才能调用 store 函数。在编写 *store* 函数时，不要忘记它是从用户空间取回信息，因此，在对其采取任何响应前，最好仔细检查其合法性。如果输入的数据与预期不符，就要返回一个负的错误码，而不是采取一些不可预期或无法恢复的行动。举例说明，假定我们的设备导出 self_destruct 属性，则应该要求必须将特定的字符串写入该属性才能调用相应的功能，而一个偶然的随机写操作应产生错误。

非默认属性

在许多情况下，kobject 类型的 default_attrs 成员描述了 kojbect 拥有的所有属性。但是在设计上，这不是一个严格的限制，我们可以根据需要对 kobject 内的属性进行添加和删除。如果希望在 kobject 的 sysfs 目录中添加新的属性，只需要填写一个 attribute 结构，并把它传递给下面的函数：

```
int sysfs_create_file(struct kobject *kobj, struct attribute *attr);
```

如果一切正常，将使用 attribute 结构中的名字创建文件，并返回 0。否则返回一个负的错误码。

请注意那些对新属性调用同一 *show()* 和 *store()* 函数以实现操作的情况。在添加一个新的非默认属性前，应采取必要的步骤以保证这些函数知道如何实现这些属性。

调用下面的函数删除属性：

```
int sysfs_remove_file(struct kobject *kobj, struct attribute *attr);
```

在调用之后，属性不再出现在 kobject 的 sysfs 入口中。需要知道的是，一个用户空间进程可能拥有一个指向属性的、打开的文件描述符，因此，在属性被删除后，*show* 和 *store* 函数依然可能被调用。

二进制属性

sysfs的约定要求所有属性都只能包含一个可读的文本格式值。也就是说，对创建一个可以处理大量二进制数据属性的需求是很少发生的。但是，当我们在用户空间和设备之间传递不可改变的数据时，有可能产生这种需求。比如向设备上载固件时就需要这样的功能。如果我们在系统中遇到这样的设备，就可以运行用户空间程序（通过热插拔机制），这些程序使用二进制的sysfs属性将固件代码传递给内核，其过程将在“内核固件接口”一节中讲述。

我们可以用bin_attribute结构描述二进制属性：

```
struct bin_attribute {
    struct attribute attr;
    size_t size;
    ssize_t (*read)(struct kobject *kobj, char *buffer,
                    loff_t pos, size_t size);
    ssize_t (*write)(struct kobject *kobj, char *buffer,
                    loff_t pos, size_t size);
};
```

这里，attr是一个attribute结构，它给出了名字、所有者、二进制属性的权限。size是二进制属性的最大长度（如果没有最大值，则设置为0）。*read*和*write*函数与子系统设备驱动程序中的相应函数工作方式类似。它们可以在一次加载过程中被调用多次。每次调用所能操作的最大数据量是一页。sysfs中没有方法可以通知最后一个写操作已经完成，因此实现二进制属性操作的代码必须能用其他方法判断是否已经操作到数据的末尾。

必须显式创建二进制属性，也就是说，它们不能作为默认属性被设置。调用下面的函数可创建二进制属性：

```
int sysfs_create_bin_file(struct kobject *kobj,
                          struct bin_attribute *attr);
```

使用下面的函数删除二进制属性：

```
int sysfs_remove_bin_file(struct kobject *kobj,
                          struct bin_attribute *attr);
```

符号链接

sysfs文件系统具有常用的树形结构，以反映kobject之间的组织层次关系。通常内核中各对象之间的关系远比这复杂。比如一个sysfs的子树（*/sys/devices*）表示了所有系统知晓的设备。而其他的子树（在*/sys/bus*下）表示了设备的驱动程序。但是这些树并不能表示驱动程序及其所管理的设备之间的关系。为了表示这种关系还需要其他的指针，在sysfs中，通过符号链接实现了这个目的。

在sysfs中创建符号链接时使用下面的函数:

```
int sysfs_create_link(struct kobject *kobj, struct kobject *target,
                      char *name);
```

该函数创建了一个链接(称为name)指向target的sysfs入口,并作为kobj的一个属性。这是一个相对链接,因此与sysfs挂装系统中的特定位置无关。

即使target已经从文件系统中删除,该链接依然存在。如果创建指向其他kobject的符号链接,应该有某种方法探测到kobject的这些变化,或者有办法保证目标kobject不会消失。其结果(在sysfs中失效的符号链接)并不致命,但是它们并不是完美的编程风格,而且可能在用户空间引起混乱。

用下面的函数删除符号链接:

```
void sysfs_remove_link(struct kobject *kobj, char *name);
```

热插拔事件的产生

一个热插拔事件是从内核空间发送到用户空间的通知,它表明系统配置出现了变化。无论kobject被创建还是被删除,都会产生这种事件。比如,当数码相机通过USB线缆插入到系统时,或者用户切换控制台终端时,或者当给磁盘分区时,都会产生这类事件。热插拔事件会导致对*/sbin/hotplug*程序的调用,该程序通过加载驱动程序,创建设备节点,挂装分区,或者其他正确的动作来响应事件。

我们要讨论最后一个重要的kobject函数用来产生这些事件。当我们把kobject传递给*kobject_add*或者*kobject_del*时,才会真正产生这些事件。在事件被传递到用户空间之前,处理kobject(或者准确一些,是kobject所属的kset)的代码能够为用户空间添加信息,或者完全禁止事件的产生。

热插拔操作

对热插拔事件的实际控制,是由保存在kset_hotplug_ops结构中的函数完成的:

```
struct kset_hotplug_ops {
    int (*filter)(struct kset *kset, struct kobject *kobj);
    char *(*name)(struct kset *kset, struct kobject *kobj);
    int (*hotplug)(struct kset *kset, struct kobject *kobj,
                   char **envp, int num_envp, char *buffer,
                   int buffer_size);
};
```

我们可以在kset结构的hotplug_ops成员中发现指向这个结构的指针。如果在kset中不包含一个指定的kobject，内核将在分层结构中进行搜索（通过parent指针），直到找到一个包含有kset的kobject为止，然后使用这个kset的热插拔操作。

无论什么时候，当内核要为指定的kobject产生事件时，都要调用filter函数。如果*filter*返回0，将不产生事件。因此该函数给kset代码一个机会，用于决定是否向用户空间传递特定的事件。

使用该函数的一个例子是块设备子系统。在该子系统中，至少使用了三种类型的kobject，它们是磁盘、分区和请求队列。用户空间将会对磁盘或者分区的添加产生响应，但通常不会响应请求队列的变化。因此，*filter*函数只允许为kobject产生磁盘和分区事件。请看下面的代码：

```
static int block_hotplug_filter(struct kset *kset, struct kobject *kobj)
{
    struct kobj_type *ktype = get_ktype(kobj);

    return ((ktype =  = &ktype_block) || (ktype =  = &ktype_part));
}
```

这里对kobject的快速类型检查足以用来判断是否产生事件。

在调用用户空间的热插拔程序时，相关子系统的名字将作为唯一的参数传递给它。hotplug方法负责提供这个名字，它将返回一个适合传递给用户空间的字符串。

任何热插拔脚本所需要知道的信息将通过环境变量传递。最后一个hotplug方法（*hotplug*）会在调用脚本前，提供添加环境变量的机会。该方法的原型是：

```
int (*hotplug)(struct kset *kset, struct kobject *kobj,
               char **envp, int num_envp, char *buffer,
               int buffer_size);
```

和先前一样，kset和kobject描述了产生事件的目的对象。envp是一个保存其他环境变量定义的数组（一般使用NAME=value的格式），num_envp说明目前有多少个变量入口。变量应当在编码后放入长度为buffer_size的缓冲区中。如果要在envp中添加任何变量，请确保在最后一个新变量后加入NULL入口，这样内核就知道哪里是结束点了。该方法的通常返回值是0，返回任何非0值将终止热插拔事件的产生。

热插拔事件的创建（如同设备模型中的许多工作一样）通常被总线驱动程序级别上的逻辑所控制。

总线、设备和驱动程序

到目前为止，读者已经看到了许多低层程序的片段和相关的例子，在本章的剩余部分，将讲述Linux设备模型的高级部分。为达到这个目的，我们会介绍一个新的虚拟总线，称之为*lddbus*（注1），并且修改*scullp*驱动程序来“连接”到这个总线。

再强调一遍，这里讲述的大部分内容，对许多驱动程序作者来说是不必要的。通常这个层次的具体细节集中于总线层，只有很少的驱动程序作者需要添加一个新的总线类型。本章内容的读者主要是那些希望了解PCI、USB等设备的工作原理，或者是需要修改这些代码的程序员。

总线

总线是处理器与一个或者多个设备之间的通道。在设备模型中，所有的设备都通过总线相连。甚至是那些内部的虚拟“平台”总线。总线可以互相插入，比如一个USB控制器通常是一个PCI设备。设备模型展示了总线和它们所控制的设备之间的连接。

在Linux设备模型中，用bus_type结构表示总线，它的定义包含在*<linux/device.h>*中。其结构如下：

```
struct bus_type {
    char *name;
    struct subsystem subsys;
    struct kset drivers;
    struct kset devices;
    int (*match)(struct device *dev, struct device_driver *drv);
    struct device *(*add)(struct device * parent, char * bus_id);
    int (*hotplug) (struct device *dev, char **envp,
                    int num_envp, char *buffer, int buffer_size);
    /* 这里省略了一些成员 */
};
```

name成员是总线的名字，比如pci。我们可以从这个结构中看到，每个总线都有自己的子系统。然而这些子系统并不在sysfs中的顶层，相反，我们会在总线子系统下面发现它们。一个总线包含两个kset，分别代表了总线的驱动程序和插入总线的所有设备。另外还有一组方法，将在下面讲到。

注1：当然，该总线的逻辑名称应该是“sbus”，但我们还是取了一个真正的、物理总线的名字。

总线的注册

正如读者注意到的，例子源代码包含了一个叫*lddbus*的虚拟总线的实现。这个总线用下面的代码设置bus_type结构：

```
struct bus_type ldd_bus_type = {
    .name = "ldd",
    .match = ldd_match,
    .hotplug  = ldd_hotplug,
};
```

请注意，只有非常少的bus_type成员需要初始化；它们中的大多数都由设备模型核心所控制。但是，我们必须为总线指定名字以及其他一些必要的方法。

对于新的总线，我们必须调用*bus_register*进行注册。*lddbus*使用下面的代码完成注册：

```
ret = bus_register(&ldd_bus_type);
if (ret)
    return ret;
```

当然，这个调用可能会失败，因此必须检查它的返回值。如果成功，新的总线子系统将被添加到系统中，可以在sysfs的*/sys/bus*目录下看到它。然后，我们可以向这个总线添加设备。

当有必要从系统中删除一个总线的时候（比如相应的模块被删除），要使用*bus_unregister*函数：

```
void bus_unregister(struct bus_type *bus);
```

总线方法

在bus_type结构中，定义了许多方法，这些方法允许总线核心作为中间介质，在设备核心与单独的驱动程序之间提供服务。2.6.10内核定义的总线方法有：

`int (*match)(struct device *device, struct device_driver *driver);`

当一个总线上的新设备或者新驱动程序被添加时，会一次或多次调用这个函数。如果指定的驱动程序能够处理指定的设备，该函数返回非零值（不久将详细讲述device和device_driver结构）。必须在总线层上使用该函数，因为那里存在着正确的逻辑。核心内核不知道如何为每个总线类型匹配设备和驱动程序。

`int (*hotplug) (struct device *device, char **envp, int num_envp, char *buffer, int buffer_size);`

在为用户空间产生热插拔事件前，这个方法允许总线添加环境变量。其参数与kset的*hotplug*方法相同（在前面的“热插拔事件的产生”一节中讲述）。

lddbus驱动程序有一个非常简单的match方法，它只是简单地比较了驱动程序和设备的名字：

```
static int ldd_match(struct device *dev, struct device_driver *driver)
{
    return !strncmp(dev->bus_id, driver->name, strlen(driver->name));
}
```

在调用真实的硬件时，*match* 函数通常对设备提供的硬件ID和驱动所支持的ID做某种类型的比较。

下面是 *lddbus* 的 *hotplug* 函数：

```
static int ldd_hotplug(struct device *dev, char **envp, int num_envp,
        char *buffer, int buffer_size)
{
    envp[0] = buffer;
    if (snprintf(buffer, buffer_size, "LDDBUS_VERSION=%s",
                Version) >= buffer_size)
        return -ENOMEM;
    envp[1] = NULL;
    return 0;
}
```

这里，我们只是在环境变量中添加了*lddbus*源代码的当前版本号，以便读者做相应的了解。

对设备和驱动程序的迭代

如果要编写总线层代码，可能会发现不得不对注册到总线的所有设备和驱动程序执行某些操作。这可能需要仔细研究嵌入到bus_type结构中的其他数据结构，但是使用内核提供的辅助函数会更好一些。

为了操作注册到总线的每个设备，可使用：

```
int bus_for_each_dev(struct bus_type *bus, struct device *start,
                     void *data, int (*fn)(struct device *, void *));
```

该函数迭代了在总线上的每个设备，将相关的device结构传递给fn，同时传递data值。如果start是NULL，将从总线上的第一个设备开始迭代；否则将从start后的第一个设备开始迭代。如果fn返回一个非零值，将停止迭代，而这个值也会从*bus_for_each_dev*返回。

相似的函数也可用于驱动程序的迭代上：

```
int bus_for_each_drv(struct bus_type *bus, struct device_driver *start,
                     void *data, int (*fn)(struct device_driver *, void *));
```

该函数的工作方式与 *bus_for_each_dev* 相同，只是它的工作对象是驱动程序而已。

值得注意的是，这两个函数在工作期间，都会拥有总线子系统的读取者/写入者信号量。因此，同时使用这两个函数会发生死锁 —— 它们中的任何一个函数都试图获得相同的信号量。修改总线的操作（比如注销设备）也有同样的问题。因此使用 *bus_for_each* 函数要多加小心。

总线属性

几乎在 Linux 设备模型的每一层都提供了添加属性的函数，总线层也不例外。bus_attribute 类型在 *<linux/device.h>* 中定义，其代码如下：

```
struct bus_attribute {
    struct attribute attr;
    ssize_t (*show)(struct bus_type *bus, char *buf);
    ssize_t (*store)(struct bus_type *bus, const char *buf,
                     size_t count);
};
```

已经在“默认属性”一节中讨论过 attribute 结构了。bus_attribute 类型也包括了两个用来显示和设置属性值的函数。大多数在 kobject 级以上的设备模型层都是按此种方式工作的。

有一个非常便于使用的宏，可在编译时刻创建和初始化 bus_attribute 结构：

```
BUS_ATTR(name, mode, show, store);
```

这个宏声明了一个结构，它将 bus_attr_ 作为给定 name 的前缀来创建总线的真正名称。

创建属于总线的任何属性，都需要显式调用 *bus_create_file* 函数：

```
int bus_create_file(struct bus_type *bus, struct bus_attribute *attr);
```

也可以使用下面的函数删除属性：

```
void bus_remove_file(struct bus_type *bus, struct bus_attribute *attr);
```

lddbus 驱动程序创建了一个包含版本号的属性文件。其中 *show* 函数和 bus_attribute 结构使用下面的代码设置：

```
static ssize_t show_bus_version(struct bus_type *bus, char *buf)
{
    return snprintf(buf, PAGE_SIZE, "%s\n", Version);
}

static BUS_ATTR(version, S_IRUGO, show_bus_version, NULL);
```

在模块的装载阶段创建属性文件:

```
if (bus_create_file(&ldd_bus_type, &bus_attr_version))
    printk(KERN_NOTICE "Unable to create version attribute\n");
```

在 *lddbus* 中，上面的语句创建了一个包含版本号的属性文件（*/sys/bus/ldd/version*）。

设备

在最底层，Linux 系统中的每一个设备都用 device 结构的一个实例来表示:

```
struct device {
    struct device *parent;
    struct kobject kobj;
    char bus_id[BUS_ID_SIZE];
    struct bus_type *bus;
    struct device_driver *driver;
    void *driver_data;
    void (*release)(struct device *dev);
    /* 省略了几个成员 */
};
```

在device结构中还有许多包含其他结构的成员，它们只对设备核心代码起重要的作用。但是了解以下这些成员是非常值得的:

`struct device *parent`

设备的“父”设备 —— 指的是该设备所属的设备。在大多数情况下，一个父设备通常是某种总线或者是宿主控制器。如果parent是NULL，表示该设备是顶层设备，但这种情况很少出现。

`struct kobject kobj;`

表示该设备并把它连接到结构体系中的 kobject。请注意，作为一个通用准则，device->kobj->parent 与 &device->parent->kobj 是相同的。

`char bus_id[BUS_ID_SIZE];`

在总线上唯一标识该设备的字符串。比如 PCI 设备使用了标准 PCI ID 格式，它包括：域编号、总线编号、设备编号和功能编号。

`struct bus_type *bus;`

标识了该设备连接在何种类型的总线上。

`struct device_driver *driver;`

管理该设备的驱动程序。在下一节中将介绍 device_driver 结构。

`void *driver_data;`

由设备驱动程序使用的私有数据成员。

void (*release)(struct device *dev);

当指向设备的最后一个引用被删除时，内核调用该方法；它将从内嵌的kobject的*release*方法中调用。所有向核心注册的device结构都必须有一个*release*方法，否则内核将打印出错误信息。

在注册device结构前，至少要设置parent、bus_id、bus和release成员。

设备注册

常用的注册和注销函数是：

```
int device_register(struct device *dev);
void device_unregister(struct device *dev);
```

我们已经看到了*lddbus*代码是如何注册它的总线类型的，然而，一个实际的总线是一个设备，因此必须被单独注册。出于简化的原因，*lddbus*模块只支持了单独的虚拟总线，因此，驱动程序在编译时构造它的设备：

```
static void ldd_bus_release(struct device *dev)
{
    printk(KERN_DEBUG "lddbus release\n");
}

struct device ldd_bus = {
    .bus_id   = "ldd0",
    .release  = ldd_bus_release
};
```

这是一个顶层总线，因此parent和bus成员是NULL，而*release*方法不做任何实质性的工作。作为第一个（也是唯一一个）总线，它的名字是ldd0。该总线用下面的函数注册：

```
ret = device_register(&ldd_bus);
if (ret)
    printk(KERN_NOTICE "Unable to register ldd0\n");
```

完成这个调用后，我们就可以在sysfs中的*/sys/devices*目录中看到它。任何添加到该总线的设备都会在*/sys/devices/ldd0/*中显示。

设备属性

sysfs中的设备入口可以有属性。相关的结构是：

```
struct device_attribute {
    struct attribute attr;
    ssize_t (*show)(struct device *dev, char *buf);
```

```
        ssize_t (*store)(struct device *dev, const char *buf,
                         size_t count);
};
```

我们可以在编译时刻用下面的宏构造这些attribute结构:

```
DEVICE_ATTR(name, mode, show, store);
```

该结构将dev_attr_作为指定名字的前缀来构造结构的名称。用下面的两个函数实现对属性文件的实际处理:

```
int device_create_file(struct device *device,
                       struct device_attribute *entry);
void device_remove_file(struct device *dev,
                        struct device_attribute *attr);
```

bus_type结构中的dev_attrs成员，指向一个为每个加入总线的设备建立的默认属性链表。

设备结构的嵌入

device结构中包含了设备模型核心用来模拟系统的信息。然而，大多数子系统记录了它们所拥有设备的其他信息，因此，单纯用device结构表示的设备是很少见的，而是通常把类似kobject这样的结构内嵌在设备的高层表示之中。如果读者阅读pci_dev或者usb_device结构的定义，就会发现其中隐藏了device结构。通常，底层驱动程序并不知道device结构，但是也有例外。

*lddbus*驱动程序创建了自己的device类型（ldd_device结构），并希望每个设备驱动程序使用这个类型注册它们的设备。这是一个简单的结构:

```
struct ldd_device {
    char *name;
    struct ldd_driver *driver;
    struct device dev;
};

#define to_ldd_device(dev) container_of(dev, struct ldd_device, dev);
```

该结构允许驱动程序为设备提供实际的名字(它与保存在device结构中的总线ID不同)，还提供一个指向驱动程序信息的指针。真实设备的结构通常包含供应商信息、设备模型、设备配置、使用的资源等其他信息。pci_dev (*<linux/pci>*) 结构或者usb_device (*<linux/usb.h>*) 结构都是非常好的例子。我们为ldd_device定义了一个宏(*to_ldd_device*)，以便于将嵌入的device结构指针转化为ldd_device指针。

*lddbus*导出的注册接口如下:

```
int register_ldd_device(struct ldd_device *lddddev)
{
    lddddev->dev.bus = &ldd_bus_type;
    lddddev->dev.parent = &ldd_bus;
    lddddev->dev.release = ldd_dev_release;
    strncpy(lddddev->dev.bus_id, lddddev->name, BUS_ID_SIZE);
    return device_register(&lddddev->dev);
}
EXPORT_SYMBOL(register_ldd_device);
```

如上所示，只是简单地填充了嵌入的device结构中一些成员（单独的驱动程序没有必要知道这些），并且向驱动程序核心注册设备。我们也可以在这里添加总线专有的设备属性。

为了展示这个接口是如何被使用的，现在向读者介绍另外一个例子驱动程序，称之为*sculld*。它是先前在第八章介绍的*scullp*驱动程序的另外一个版本。它实现了通常的内存区域设备，但是*sculld*可通过*lddbus*接口利用Linux设备模型工作。

*sculld*驱动程序向它的设备入口添加了一个自己的属性，称之为dev，它只包含了相关的设备编号。这个属性可以由模块装载脚本，或者热插拔子系统使用，以便在设备添加到系统中时自动创建设备节点。该属性的设置使用以下代码：

```
static ssize_t sculld_show_dev(struct device *ddev, char *buf)
{
    struct sculld_dev *dev = ddev->driver_data;

    return print_dev_t(buf, dev->cdev.dev);
}

static DEVICE_ATTR(dev, S_IRUGO, sculld_show_dev, NULL);
```

然后在初始化时注册设备，并且用下面的函数创建dev属性：

```
static void sculld_register_dev(struct sculld_dev *dev, int index)
{
    sprintf(dev->devname, "sculld%d", index);
    dev->ldev.name = dev->devname;
    dev->ldev.driver = &sculld_driver;
    dev->ldev.dev.driver_data = dev;
    register_ldd_device(&dev->ldev);
    device_create_file(&dev->ldev.dev, &dev_attr_dev);
}
```

请注意这里使用了driver_data成员保存了指向自身内部device结构的指针。

设备驱动程序

设备模型跟踪所有系统所知道的设备。进行跟踪的主要原因是让驱动程序核心协调驱动

程序与新设备之间的关系。一旦驱动程序是系统中的已知对象，就可能完成大量的工作。例如，设备驱动程序可以导出信息和配置变量，而这些东西都是独立于任何特定设备的。

驱动程序由以下结构定义：

```
struct device_driver {
    char *name;
    struct bus_type *bus;
    struct kobject kobj;
    struct list_head devices;
    int (*probe)(struct device *dev);
    int (*remove)(struct device *dev);
    void (*shutdown) (struct device *dev);
};
```

再次强调，结构中的许多成员已经被忽略掉了（请参看*<linux/device.h>*了解全部的成员）。其中，name是驱动程序的名字（它将在sysfs中显示），bus是该驱动程序所操作的总线类型，kobj是必需的kobject，devices是当前驱动程序能操作的设备链表，*probe*是用来查询特定设备是否存在的函数（以及这个驱动程序是否能操作它），当设备从系统中删除的时候要调用remove函数，在关机的时候调用shutdown函数关闭设备。

操作device_driver结构的函数形式现在看起来会很熟悉（将很快讨论它们）。它的注册函数是：

```
int driver_register(struct device_driver *drv);
void driver_unregister(struct device_driver *drv);
```

常用的属性结构是：

```
struct driver_attribute {
    struct attribute attr;
    ssize_t (*show)(struct device_driver *drv, char *buf);
    ssize_t (*store)(struct device_driver *drv, const char *buf,
                     size_t count);
};
DRIVER_ATTR(name, mode, show, store);
```

使用下面的函数创建属性文件：

```
int driver_create_file(struct device_driver *drv,
                       struct driver_attribute *attr);
void driver_remove_file(struct device_driver *drv,
                        struct driver_attribute *attr);
```

bus_type结构包含了一个成员（drv_attrs），它指向一组为属于该总线的所有设备创建的默认属性。

驱动程序结构的嵌入

对于大多数驱动程序核心结构来说，device_driver结构通常被包含在高层和总线相关的结构中。*lddbus*子系统也不违反这一原则，因此它定义了自己的ldd_driver结构：

```
struct ldd_driver {
    char *version;
    struct module *module;
    struct device_driver driver;
    struct driver_attribute version_attr;
};

#define to_ldd_driver(drv) container_of(drv, struct ldd_driver, driver);
```

这里，我们要求每个驱动程序提供自己当前的软件版本号，*lddbus*为它所知道的每一个驱动程序导出这个版本字符串。该总线特有的驱动程序注册函数如下：

```
int register_ldd_driver(struct ldd_driver *driver)
{
    int ret;

    driver->driver.bus = &ldd_bus_type;
    ret = driver_register(&driver->driver);
    if (ret)
        return ret;
    driver->version_attr.attr.name = "version";
    driver->version_attr.attr.owner = driver->module;
    driver->version_attr.attr.mode = S_IRUGO;
    driver->version_attr.show = show_version;
    driver->version_attr.store = NULL;
    return driver_create_file(&driver->driver, &driver->version_attr);
}
```

该函数的前半部分只是简单地向核心注册了低层的device_driver结构，其余部分设置了版本号属性。由于该属性是在运行时建立的，因此不能使用DRIVER_ATTR宏，这样，我们必须手工填写driver_attribute结构。请注意要将owner属性设置为驱动程序模块，而不是*lddbus*模块，这么做的原因可以在为该属性实现的*show*函数中看到：

```
static ssize_t show_version(struct device_driver *driver, char *buf)
{
    struct ldd_driver *ldriver = to_ldd_driver(driver);

    sprintf(buf, "%s\n", ldriver->version);
    return strlen(buf);
}
```

也许有的读者会想，owner属性应该是*lddbus*模块，因为在那里定义了实现该属性的函数。然而，该函数使用驱动程序自己创建和拥有的ldd_driver结构。如果该结构已经不存在了，而用户空间进程又要试图读取版本号，则可能会出现麻烦。将owner属性

设置为驱动程序模块可防止用户空间打开属性文件时卸载模块的情况发生。由于每个驱动程序模块创建了 *lddbus* 模块的引用，因此可以保证不会在不适当的时候被卸载。

考虑到完整性，*sculld* 用下面的代码创建自己的 `ldd_driver` 结构：

```
static struct ldd_driver sculld_driver = {
    .version = "$Revision: 1.1 $",
    .module = THIS_MODULE,
    .driver = {
        .name = "sculld",
    },
};
```

一个 *register_ldd_driver* 调用将它添加到了系统中。一旦初始化完成，就可以在sysfs中看到驱动程序信息：

```
$ tree /sys/bus/ldd/drivers
/sys/bus/ldd/drivers
`-- sculld
    |-- sculld0 -> ../../../../devices/ldd0/sculld0
    |-- sculld1 -> ../../../../devices/ldd0/sculld1
    |-- sculld2 -> ../../../../devices/ldd0/sculld2
    |-- sculld3 -> ../../../../devices/ldd0/sculld3
    `-- version
```

类

本章讨论的最后一个设备模型概念是类。类是一个设备的高层视图，它抽象出了低层的实现细节。驱动程序看到的是SCSI磁盘和ATA磁盘，但是在类的层次上，它们都是磁盘而已。类允许用户空间使用设备所提供的功能，而不关心设备是如何连接的，以及它们是如何工作的。

几乎所有的类都显示在 */sys/class* 目录中。举个例子，不管网络接口的类型是什么，所有的网络接口都集中在 */sys/class/net* 下。输入设备可以在 */sys/class/input* 下找到，而串行设备都集中在 */sys/class/tty* 中。一个例外是块设备，出于历史的原因，它们出现在 */sys/block* 下。

类成员通常被上层代码所控制，而不需要来自驱动程序的明确支持。当 *sbull* 驱动程序（参看第十六章）创建一个虚拟磁盘设备时，它将自动出现在 */sys/block* 中。*snull* 网络驱动程序（参看第十七章）也不必为 */sys/class/net* 中出现的接口做任何特殊的事情。但是，有些情况下驱动程序也需要直接处理类。

在许多情况下，类子系统是向用户空间导出信息的最好方法。当子系统创建一个类时，它将完全拥有这个类，因此根本不必担心哪个模块拥有那些属性。只要花很少的时间观

察一下 sysfs 中那些面向硬件的部分，就会发现它的表示并不十分友好，我们会更愿意在*/sys/class/*下查找信息，而不是在*/sys/devices/pci0000:00/0000:00:10.0/usb2/2-0:1.0*中查找。

为了管理类，驱动程序核心导出了两个不同的接口。*class_simple*例程提供了一种尽可能简单的方法，来向系统中添加新的类；通常这些例程的主要目的是，提供包含设备号的属性以便创建设备节点。正规的类的接口更复杂一些，但也提供了更多的功能。这里从简单的接口入手学习。

class_simple 接口

*class_simple*接口非常易于使用，甚至用不着用户担心导出包含已分配设备号属性这样的信息。这个接口只是一些简单的函数调用，几乎没有使用Linux设备模型的样板文件。

第一步是创建类本身。调用*class_simple_create*函数完成这一任务：

```
struct class_simple *class_simple_create(struct module *owner, char *name);
```

该函数使用给定的名字创建类。这个操作有可能会失败，因此在进行下一步前，应始终检查它的返回值（使用第十一章“指针和错误值”一节中介绍的*IS_ERR*）。

可以用下面的函数销毁一个简单类：

```
void class_simple_destroy(struct class_simple *cs);
```

创建一个简单类的真实目的是为它添加设备，我们可使用下面的函数达到这一目的：

```
struct class_device *class_simple_device_add(struct class_simple *cs,
                                             dev_t devnum,
                                             struct device *device,
                                             const char *fmt, ...);
```

这里，cs是前面创建的简单类，devnum是分配的设备号，device是表示这个设备的device结构，剩下的参数是用来创建设备名称的、*printk*风格的格式字符串和参数。该调用向包含设备号属性（dev）的类中添加了一个入口。如果device参数不是NULL，一个符号链接（称为device）将指向*/sys/devices*下的设备入口。

可以向设备入口添加其他属性，这时可使用*class_device_create_file*函数，该函数和类子系统的其余部分将在下一节讲述。

在插拔设备时，类会产生热插拔事件。如果驱动程序需要为用户空间处理程序添加环境变量的话，可以用下面的代码设置热插拔回调函数：

```
int class_simple_set_hotplug(struct class_simple *cs,
                             int (*hotplug)(struct class_device *dev,
                                            char **envp, int num_envp,
                                            char *buffer, int buffer_size));
```

当拔除设备时，使用下面的函数删除类入口：

```
void class_simple_device_remove(dev_t dev);
```

请注意，这里并不需要*class_simple_device_add*返回的class_device结构，提供设备号（应该是唯一的）就足够了。

完整的类接口

*class_simple*接口能够满足许多需求，但有时候需要有更强的灵活性。下面的讨论将描述如何使用基于*class_simple*的完整类机制。简而言之，类函数和结构与设备模型的其他部分遵从相同的模式，因此真正崭新的概念是很少的。

管理类

用class结构的一个实例来定义类：

```
struct class {
    char *name;
    struct class_attribute *class_attrs;
    struct class_device_attribute *class_dev_attrs;
    int (*hotplug)(struct class_device *dev, char **envp,
                   int num_envp, char *buffer, int buffer_size);
    void (*release)(struct class_device *dev);
    void (*class_release)(struct class *class);
    /* 省略了一些成员 */
};
```

每个类都需要一个唯一的名字，它将显示在*/sys/class*中。一个类被注册后，将创建class_attrs指向的数组（以NULL结尾）中的所有属性。还要添加每个设备的一组默认属性，而class_dev_attrs指针指向了这些属性。当热插拔事件产生时，还有一个常用的*hotplug*函数用来添加环境变量。还有另外两个*release*方法：把设备从类中删除时，调用*release*方法，而释放类本身时，调用*class_release*方法。

注册函数是：

```
int class_register(struct class *cls);
void class_unregister(struct class *cls);
```

处理属性的接口对读者来说已经没有什么稀奇的了：

```
struct class_attribute {
    struct attribute attr;
    ssize_t (*show)(struct class *cls, char *buf);
    ssize_t (*store)(struct class *cls, const char *buf, size_t count);
};

CLASS_ATTR(name, mode, show, store);
int class_create_file(struct class *cls,
                      const struct class_attribute *attr);
void class_remove_file(struct class *cls,
                       const struct class_attribute *attr);
```

类设备

类存在的真正目的是，给作为类成员的各个设备提供一个容器。用class_device结构表示类的成员：

```
struct class_device {
    struct kobject kobj;
    struct class *class;
    struct device *dev;
    void *class_data;
    char class_id[BUS_ID_SIZE];
};
```

class_id成员包含了要在sysfs中显示的设备名。class指针指向包含该设备的类，dev指向与此相关的device结构。对dev的设置是可选的；如果它不是NULL，它将创建一个从类入口到*/sys/devices*下相应入口的符号链接，使得在用户空间查找设备入口非常简单。类使用class_data保存私有数据指针。

常用的注册函数如下：

```
int class_device_register(struct class_device *cd);
void class_device_unregister(struct class_device *cd);
```

类设备接口还允许已经注册过的入口项改名：

```
int class_device_rename(struct class_device *cd, char *new_name);
```

类设备入口具有属性：

```
struct class_device_attribute {
   struct attribute attr;
   ssize_t (*show)(struct class_device *cls, char *buf);
   ssize_t (*store)(struct class_device *cls, const char *buf,
                    size_t count);
};
CLASS_DEVICE_ATTR(name, mode, show, store);

int class_device_create_file(struct class_device *cls,
```

```
                                const struct class_device_attribute *attr);
void class_device_remove_file(struct class_device *cls,
                              const struct class_device_attribute *attr);
```

在类的class_dev_attrs成员中保存了默认的属性，在注册类设备的时候，就会创建这些属性。*class_device_create_file* 用来创建其他的属性。属性也能被添加到由 *class_simple* 接口创建的类设备中去。

类接口

类子系统具有一个在Linux设备模型其他部分找不到的附加概念。该机制被称为“接口”，但是把它理解成一种设备加入或者离开类时获得信息的触发机制更为贴切些。

一个接口由下面的结构表达：

```
struct class_interface {
    struct class *class;
    int (*add) (struct class_device *cd);
    void (*remove) (struct class_device *cd);
};
```

可以用下面的函数注册和注销接口：

```
int class_interface_register(struct class_interface *intf);
void class_interface_unregister(struct class_interface *intf);
```

接口函数都是望其名知其意的。无论何时把一个类设备添加到class_interface结构所指定的类中，都将调用接口的 *add* 函数。该函数能为设备做一些其他的必要设置，通常这些设置表现为添加更多的属性，但也能完成其他一些工作。在从类中删除设备时，调用 *remove* 函数来做必要的清理工作。

可以为单个类注册多个接口。

各环节的整合

为了能更好地理解什么是驱动程序模型，现在进入内核看一看设备生命周期的各个环节。前面讲述了PCI子系统是如何与驱动程序模型交互的，还讲解了在系统内一个驱动程序的添加与删除，以及一个设备的添加与删除的概念。这些细节虽然是针对PCI核心代码讲解的，但是也适用于其他所有使用驱动程序核心管理驱动程序和设备的子系统。

在PCI核心、驱动程序核心以及单独的PCI驱动程序之间的交互是非常复杂的，如图14-3所示。

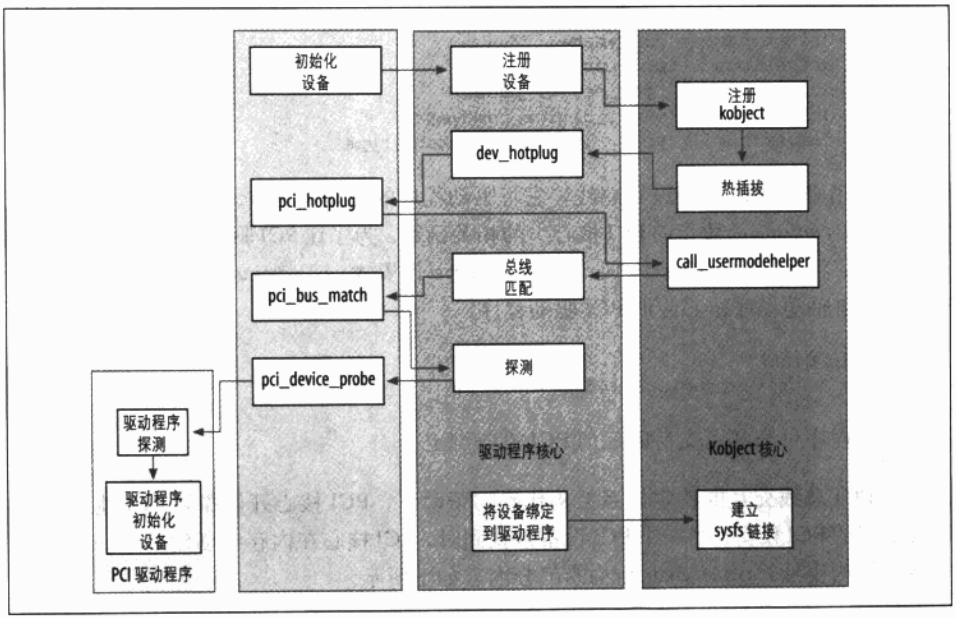


图 14-3: 设备创建过程

添加一个设备

PCI 子系统声明了一个 bus_type 结构，称为 pci_bus_type，它由下面的值初始化:

```
struct bus_type pci_bus_type = {
    .name       = "pci",
    .match      = pci_bus_match,
    .hotplug    = pci_hotplug,
    .suspend    = pci_device_suspend,
    .resume     = pci_device_resume,
    .dev_attrs = pci_dev_attrs,
};
```

在将PCI子系统装载到内核中时，通过调用*bus_register*，该pci_bus_type变量将向驱动程序核心注册。此后，驱动程序将在*/sys/bus/pci*中创建一个sysfs目录，其中包含了两个目录: *devices*和*drivers*。

所有的PCI驱动程序都必须定义一个pci_driver结构变量，在该变量包含了这个PCI驱动程序所提供的不同功能函数(关于PCI子系统，以及如何编写PCI驱动程序的知识，请参看第十二章)。这个结构中包含了一个device_driver结构，在注册PCI驱动程序时，这个结构将被初始化:

```
/* 初始化driver中常用的成员 */
drv->driver.name = drv->name;
drv->driver.bus = &pci_bus_type;
drv->driver.probe = pci_device_probe;
drv->driver.remove = pci_device_remove;
drv->driver.kobj.ktype = &pci_driver_kobj_type;
```

该段代码用来为驱动程序设置总线，它将驱动程序的总线指向pci_bus_type，并且将*probe*和*remove*函数指向PCI核心中的相关函数。为了让PCI驱动程序的属性文件能正常工作，将驱动程序的kobject中的ktype设置成pci_driver_kobj_type。然后PCI核心向驱动程序核心注册PCI驱动程序:

```
/* 向核心注册 */
error = driver_register(&drv->driver);
```

现在驱动程序可与其所支持的任何PCI设备绑定。

在能与PCI总线交互的特定体系架构代码的帮助下，PCI核心开始探测PCI地址空间，查找所有的PCI设备。当一个PCI设备被找到时，PCI核心在内存中创建一个pci_dev类型的结构变量。pci_dev结构的部分内容如下所示:

```
struct pci_dev {
    /* ... */
    unsigned int    devfn;
    unsigned short vendor;
    unsigned short device;
    unsigned short subsystem_vendor;
    unsigned short subsystem_device;
    unsigned int    class;
    /* ... */
    struct pci_driver *driver;
    /* ... */
    struct device dev;
    /* ... */
};
```

这个PCI设备中与总线相关的成员将由PCI核心初始化（devfn、vendor、device以及其他成员），并且device结构变量的parent变量被设置为该PCI设备所在的总线设备。bus变量被设置为指向pci_bus_type结构。接着设置name和bus_id变量，其值取决于从PCI设备中读取的名字和ID。

当PCI的device结构被初始化后，使用下面的代码向驱动程序核心注册设备:

```
device_register(&dev->dev);
```

在*device_register*函数中，驱动程序核心对device中的许多成员进行初始化，向kobject核心注册设备的kobject（这将产生一个热插拔事件，在本章后面的部分讨论），然后将

该设备添加到设备列表中，该设备列表为包含该设备的父节点所拥有。完成这些工作后，所有的设备都可以通过正确的顺序访问，并且知道每个设备都挂在层次结构的哪一点上。

接着设备将被添加到与总线相关的所有设备链表中，在这个例子中是pci_bus_type链表。这个链表包含了所有向总线注册的设备，遍历这个链表，并且为每个驱动程序调用该总线的*match*函数，同时指定该设备。对于*pci_bus_type*总线来说，PCI核心在把设备提交给驱动程序核心前，将*match*函数指向*pci_bus_match*函数。

*pci_bus_match*函数将把驱动程序核心传递给它的device结构转换为pci_dev结构。它还把device_driver结构转换为pci_driver结构，并且查看设备和驱动程序中的PCI设备相关信息，以确定驱动程序是否能支持这类设备。如果这样的匹配工作没能正确执行，该函数会向驱动程序核心返回0，接着驱动程序核心考虑在其链表中的下一个驱动程序。

如果匹配工作圆满完成，函数向驱动程序核心返回1。这将导致驱动程序核心将device结构中的driver指针指向这个驱动程序，然后调用device_driver结构中指定的*probe*函数。

在PCI驱动程序向驱动程序核心注册前，probe变量被设置为指向*pci_device_probe*函数。该函数将device结构转换为pci_dev结构，并且把在device中设置的driver结构转换为pci_driver结构。它也将检测这个驱动程序的状态，以确保其能支持这个设备（出于某种原因，这个检测有点多余），增加设备的引用计数，然后用绑定的pci_dev结构指针为参数，调用PCI驱动程序的*probe*函数。

如果PCI驱动程序的*probe*函数出于某种原因，判定不能处理这个设备，其将返回负的错误值给驱动程序核心，这将导致驱动程序核心继续在驱动程序列表中搜索，以匹配这个设备。如果*probe*函数探测到了设备，为了能正常操作设备，它将做所有的初始化工作，然后向驱动程序核心返回0。这会使驱动程序核心将该设备添加到与此驱动程序绑定的设备链表中，并且在sysfs中的drivers目录到当前控制设备之间建立符号链接。这个符号链接使得用户知道哪个驱动程序被绑定到了哪个设备上。该目录有类似以下的内容：

```
$ tree /sys/bus/pci
/sys/bus/pci/
|-- devices
|   |-- 0000:00:00.0 -> ../../../devices/pci0000:00/0000:00:00.0
|   |-- 0000:00:00.1 -> ../../../devices/pci0000:00/0000:00:00.1
|   |-- 0000:00:00.2 -> ../../../devices/pci0000:00/0000:00:00.2
|   |-- 0000:00:02.0 -> ../../../devices/pci0000:00/0000:00:02.0
|   |-- 0000:00:04.0 -> ../../../devices/pci0000:00/0000:00:04.0
|   |-- 0000:00:06.0 -> ../../../devices/pci0000:00/0000:00:06.0
```

```
|   |-- 0000:00:07.0 -> ../../../devices/pci0000:00/0000:00:07.0
|   |-- 0000:00:09.0 -> ../../../devices/pci0000:00/0000:00:09.0
|   |-- 0000:00:09.1 -> ../../../devices/pci0000:00/0000:00:09.1
|   |-- 0000:00:09.2 -> ../../../devices/pci0000:00/0000:00:09.2
|   |-- 0000:00:0c.0 -> ../../../devices/pci0000:00/0000:00:0c.0
|   |-- 0000:00:0f.0 -> ../../../devices/pci0000:00/0000:00:0f.0
|   |-- 0000:00:10.0 -> ../../../devices/pci0000:00/0000:00:10.0
|   |-- 0000:00:12.0 -> ../../../devices/pci0000:00/0000:00:12.0
|   |-- 0000:00:13.0 -> ../../../devices/pci0000:00/0000:00:13.0
|   `-- 0000:00:14.0 -> ../../../devices/pci0000:00/0000:00:14.0
`-- drivers
    |-- ALI15x3_IDE
    |   `-- 0000:00:0f.0 -> ../../../../devices/pci0000:00/0000:00:0f.0
    |-- ehci_hcd
    |   `-- 0000:00:09.2 -> ../../../../devices/pci0000:00/0000:00:09.2
    |-- ohci_hcd
    |   |-- 0000:00:02.0 -> ../../../../devices/pci0000:00/0000:00:02.0
    |   |-- 0000:00:09.0 -> ../../../../devices/pci0000:00/0000:00:09.0
    |   `-- 0000:00:09.1 -> ../../../../devices/pci0000:00/0000:00:09.1
    |-- orinoco_pci
    |   `-- 0000:00:12.0 -> ../../../../devices/pci0000:00/0000:00:12.0
    |-- radeonfb
    |   `-- 0000:00:14.0 -> ../../../../devices/pci0000:00/0000:00:14.0
    |-- serial
    `-- trident
        `-- 0000:00:04.0 -> ../../../../devices/pci0000:00/0000:00:04.0
```

删除设备

可以使用多种不同的方法从系统中删除PCI设备。所有的CardBus设备都是真实的PCI设备，只是它们具有不同的物理形态参数，内核PCI核心对它们不加以区分。那些允许在机器运行时添加和删除PCI设备的系统正日益流行，Linux也支持这样的操作。还有一种伪PCI热插拔驱动程序，它允许开发者测试他们的PCI驱动程序能否在系统运行时正确处理设备的删除。这个模块叫做fakephp，它可以让内核认为PCI设备被拔走，但是它不允许用户从不具备这个能力的硬件系统中真正地移除该PCI设备。读者可参看该驱动程序的相关文档，以了解如何测试PCI驱动程序的更多知识。

相对于添加设备，PCI核心对删除设备做了很少的工作。当删除一个PCI设备时，要调用*pci_remove_bus_device*函数。该函数做些PCI相关的清理工作，然后使用指向pci_dev中的device结构的指针，调用*device_unregister*函数。

在*device_unregister*函数中，驱动程序核心只是删除了从绑定设备的驱动程序（如果有设备绑定的话）到sysfs文件的符号链接，从内部设备链表中删除了该设备，并且以device结构中的kobject结构指针为参数，调用*kobject_del*函数。该函数引起了用户空间的hotplug调用，表明kobject现在从系统中删除，然后删除全部与kobject相关的sysfs文件和sysfs目录，而这些目录和文件都是kobject以前创建的。

kobject_del 函数还删除了设备的kobject引用。如果该引用是最后一个（这意味着在用户空间中，没有该设备的sysfs入口文件保持打开状态），就要调用该PCI设备的*release*函数——pci_release_dev。该函数只是释放了pci_dev结构所占用的空间。

做完这些事后，与设备相关的所有sysfs入口都被删除了，并且与设备相关的内存也被释放。至此，PCI设备被完全从系统中删除。

添加驱动程序

当调用*pci_register_driver*函数时，一个PCI驱动程序被添加到PCI核心中。与前面添加设备一节中相似，该函数只是初始化了包含在pci_driver结构中的device_driver结构。PCI核心用包含在pci_driver结构中的device_driver结构指针作为参数，在驱动程序核心内调用*driver_register*函数。

*driver_register*函数初始化了几个device_driver中的锁，然后调用*bus_add_driver*函数。该函数按以下步骤操作：

- 查找与驱动程序相关的总线。如果没有找到该总线，函数立刻返回。
- 根据驱动程序的名字以及相关的总线，创建驱动程序的sysfs目录。
- 获取总线内部的锁，接着遍历所有向总线注册的设备，然后如同添加新设备一样，为这些设备调用match函数。如果match函数成功，接着如前面章节所讲，开始绑定过程的剩余步骤。

删除驱动程序

删除驱动程序是一个简单的过程。对于PCI驱动程序来说，驱动程序调用*pci_unregister_driver*函数。该函数只是用包含在pci_driver结构中的device_driver结构作为参数，调用驱动程序核心函数*driver_unregister*。

*driver_unregister*函数通过清除在sysfs树中属于驱动程序的sysfs属性，来完成一些基本管理工作。然后它遍历所有属于该驱动程序的设备，并为其调用*release*函数。这与前面将设备从系统中删除时调用*release*函数一样。

当所有的设备与驱动程序脱离后，驱动程序代码使用了下面这两个在逻辑上有点独特的函数：

```
down(&drv->unload_sem);
up(&drv->unload_sem);
```

在返回给调用者前执行这个操作。锁住代码是因为在函数安全返回前，代码需要等待驱动程序的所有引用计数为零。模块在被卸载的时候，几乎都要调用*driver_unregister*函数作为退出的方法。只要驱动程序正在被设备引用并且等待这个锁的解开，模块就需要保留在内存中，这样，内核就能知道什么时候可以安全地把驱动程序从内存中删除掉。

热插拔

有两个不同的角度来看待热插拔。从内核角度看，热插拔是在硬件、内核、内核驱动程序之间的交互。从用户的角度看，热插拔是在内核与用户之间，通过调用*/sbin/hotplug*程序的交互。当需要通知用户内核中发生了某些类型的热插拔事件时，内核才调用该函数。

动态设备

当系统正在运行时，现在几乎所有的计算机系统都能处理设备的安装与删除，在讨论这个事实的时候，用得最多的术语就是热插拔。这与前几年的计算机系统有着非常大的不同，那时候程序员都知道，当系统启动时需要扫描所有的设备，也根本不用担心在电源关闭之前，设备会被拔走。现在随着USB、CardBus、PCMCIA、IEEE1394和PCI热插拔控制器的出现，要求Linux内核能够可靠运行，而不管运行过程中在系统中添加或者删除了什么硬件。这给设备驱动程序作者增加了很大的压力，因为他们必须要处理设备被毫无征兆地突然拔走的情况。

每种不同的总线使用不同的方法处理设备的移除。比如当一个PCI、CardBus或者PCMCIA设备从系统删除时，在驱动程序通过它的*remove*函数被告之此事发生前，会留有一段时间。在这发生之前，所有对PCI总线的读操作会返回全部的字节位。这就意味着驱动程序总是要检查它们从PCI总线那里读来的数据值，并且能够正确处理值`0xff`。

这样的例子可以在*drivers/usb/host/ehci-hcd.c*驱动程序中找到，它是一个USB 2.0（高速）控制器卡的驱动。在它的主要握手循环中，通过下面的代码来检测控制器卡是否已从系统中拔掉：

```
result = readl(ptr);
if (result == ~(u32)0)      /* 拔掉了硬件卡 */
    return -ENODEV;
```

对USB驱动程序来说，当从系统中删除绑定在USB驱动程序上的设备时，任何发送给设备的未完成操作都会失败，并出现错误`-ENODEV`。驱动程序需要识别出该错误并正确删除那些未完成的I/O操作。

热插拔设备并不仅仅局限在传统设备上，比如鼠标、键盘和网卡。现在有一些系统支持添加、删除CPU和内存条。幸运的是Linux内核能正确处理添加、删除这样核心的“系统”设备，所以单独的设备驱动程序就不用在这些事情上多虑了。

/sbin/hotplug 工具

正如在本章前面所讲述的，当用户向系统添加或者删除设备时，会产生热插拔事件。这会导致内核调用用户空间程序*/sbin/hotplug*。该程序是一个典型的bash脚本程序，只是将执行权传递给其他一系列放置在*/etc/hotplug.d/*目录树中的程序。在大多数Linux发行版中，这个脚本具有与下面类似的代码：

```
DIR="/etc/hotplug.d"
for I in "${DIR}/$1/"*.hotplug "${DIR}/"default/*.hotplug ; do
    if [ -f $I ]; then
        test -x $I && $I $1 ;
    fi
done
exit 1
```

换句话说，脚本搜索所有以*.hotplug*为后缀的程序（它们有可能是处理这种事件的），并调用它们，向它们传递大量的、由内核设置的环境变量。*/sbin/hotplug*脚本的工作细节可以在程序的注释中找到，或者参看*hotplug(8)*手册页。

正如前面提到的，当kobject被创建或者删除时调用*/sbin/hotplug*。内核会使用具有一个参数的命令行调用hotplug程序，该参数就是事件的名字。核心内核及相关子系统也设置了一系列的环境变量（下面讲述）用来描述所发生的事件。hotplug程序使用这些变量去判断在内核中发生了什么，以及对此是否要采取特定的行动。

传递给*/sbin/hotplug*的命令行参数是热插拔事件的相关名称，由分派给kobject的kset来确定。我们可以调用kset的`hotplug_ops`结构中的*name*函数来设置这个名字，相关内容在本章前面的部分讲过了。如果没有提供这个函数，或者没有调用这个函数，则名字将是kset的名字。

为*/sbin/hotplug*设置的默认环境变量是：

`ACTION`

根据操作是创建对象还是删除对象，设置字符串是“`add`”或者是“`remove`”。

`DEVPATH`

在sysfs文件系统中的目录路径，指向被创建的或者被删除的kobject。请注意sysfs文件系统的挂装点并未加入这个路径，因此需要由用户空间程序判断。

SEQNUM

热插拔事件的序号。序号是一个64位的编号，随着热插拔事件的发生，该编号也随着增长。这就使得用户空间能够按照内核创建的顺序，为热插拔事件排队，当然在用户空间程序也可以不按顺序处理。

SUBSYSTEM

与上面描述相同，是传递给命令行的字符串参数。

当系统中与总线相关的设备被添加或者删除时，许多不同总线子系统都向*/sbin/hotplug*程序添加它们自己的环境变量。这一工作通过赋予所属总线（如在“热插拔操作”一节所讲）的kset_hotplug_ops结构中指定完成这一工作的*hotplug*回调函数完成。这就使得当总线发现设备时，用户空间能够自动调用任何能控制该设备的模块。下面是一个不同类型总线的清单，以及添加给*/sbin/hotplug*调用的环境变量。

IEEE1394 (FireWire)

在IEEE1394总线 —— 也称之为“火线（FireWire）”上的任何设备，将*/sbin/hotplug*的参数名和SUBSYSTEM环境变量设置为ieee1394。ieee1394子系统总是添加下面五个环境变量：

VENDOR_ID

IEEE1394设备的24位厂商ID。

MODEL_ID

IEEE1394设备的24位型号ID。

GUID

设备的64位GUID。

SPECIFIER_ID

指定设备协议所有者的24位ID。

VERSION

指定设备协议版本号的值。

网络

当网络设备在内核中注册和注销时，所有的网络设备都产生热插拔事件。*/sbin/hotplug*程序将参数名和SUBSYSTEM环境变量设置为net，并且添加下面的环境变量：

INTERFACE

向内核注册或者注销的接口名称。该值的例子有lo和eth0。

PCI

在PCI总线上的所有设备将参数名和SUBSYSTEM环境变量设置为pci。PCI子系统添加下面四个环境变量:

PCI_CLASS
: 以十六进制表示的PCI类号。

PCI_ID
: 以十六进制表示的PCI厂商和设备ID，按照vendor:device形式组合。

PCI_SUBSYS_ID
: PCI子系统厂商和子系统设备ID，按照subsys_vendor:subsys_device形式组合。

PCI_SLOT_NAME
: 内核给设备的PCI槽的名字。它以domain:bus:slot:function的方式组合，比如0000:00:0d.0。

输入

对于所有的输入设备（鼠标、键盘、游戏杆），当设备从内核中添加或删除时，都要产生热插拔事件。*/sbin/hotplug* 参数和SUBSYSTEM环境变量都设置为input。输入子系统都会添加下面的环境变量:

PRODUCT
: 一个用十六进制表示、没有前置零的多值字符串列表值。它有如下的格式:bustype:vendor:product:version。

如果设备支持，会有下面的环境变量:

NAME
: 设备给出的输入设备名。

PHYS
: 输入子系统分配给设备的物理地址。它被认为是固定的，这将取决于设备插入的总线位置。

EV

KEY

REL

ABS

MSC

LED

SND

FF

它们都是输入设备描述符，如果输入设备支持，将设置正确的值。

USB

任何在USB总线上的设备，都将把参数名和SUBSYSTEM环境变量设置为usb。USB子系统也总是添加下面的环境变量：

PRODUCT

以idVendor/idProduct/bcdDevice格式表示的字符串，用来指定这些USB的设备相关成员。

TYPE

以bDeviceClass/bDeviceSubClass/bDeviceProtocol格式表示的字符串，用来指定这些USB的设备相关成员。

如果bDeviceClass成员设置为0，则还要设置下面的环境变量：

INTERFACE

以bInterfaceClass/bInterfaceSubClass/bInterfaceProtocol格式表示的字符串，用来指定这些USB的设备相关成员。

如果使用了内核编译选项CONFIG_USB_DEVICEFS，就会将usbfs文件系统编译进内核，则还需要设置下面的环境变量：

DEVICE

显示设备在usbfs文件系统中位置的字符串。该字符串使用/proc/bus/usb/USB_BUS_NUMBER/USB_DEVICE_NUMBER格式，其中USB_BUS_NUMBER是拥有设备的USB总线的三位数号码，USB_DEVICE_NUMBER是内核分配给USB设备的三位数号码。

SCSI

当从内核中创建或者删除任何SCSI设备的时候，设备会产生一个热插拔事件。调用*/sbin/hotplug*程序的参数名称以及SUBSYSTEM环境变量设置为scsi。SCSI系统不再添加其他的环境变量，但这里提请读者注意的是，有一个SCSI相关的用户空间脚本，用来判断为这个SCSI设备装载什么样的SCSI驱动程序（磁盘、磁带、通用等等）。

便携对接站

如果从正在运行的Linux系统中添加或者删除一个支持即插即用的便携对接站（将笔记本插入或者拔出对接站），会产生一个热插拔事件。调用 */sbin/hotplug* 程序的参数名称以及 SUBSYSTEM 环境变量设置为 dock。除此之外，不会设置其他环境变量。

S/390 和 zSeries

在S/390体系架构中，通道总线体系架构支持大量的硬件设备，当从Linux虚拟系统中添加或者删除这些设备时，都能产生 */sbin/hotplug* 事件。这些设备将 */sbin/hotplug* 的参数名称以及 SUBSYSTEM 环境变量设置为 dasd。除此之外，不会设置其他环境变量。

使用 /sbin/hotplug

现在，当向内核添加或者删除任何设备时，Linux 内核将调用 */sbin/hotplug*，为此，用户空间存在许多有用的工具以便利用这一特性。其中两个最有用的工具是Linux Hotplug（热插拔）脚本和 *udev*。

Linux 热插拔脚本

Linux 热插拔脚本作为 */sbin/hotplug* 程序的第一个用户而启动。这些脚本在内核中搜索那些为描述设备而设置的不同的环境变量，从而发现设备并为其找到与之匹配的内核模块。

正如前面所述，当一个驱动程序使用 MODULE_DEVICE_TABLE 宏时，depmod 程序使用这些信息并创建了 */lib/module/KERNEL_VERSION/modules.*map* 文件。“*”将根据驱动程序所支持总线的不同而不同。现在为驱动程序而生成的模块映像文件可以很好地支持 PCI、USB、IEEE1394、INPUT、ISAPNP 和 CCW 子系统了。

热插拔脚本使用这些模块的映像文本文件来判断，为支持内核最新发现的设备要装载什么模块。它们加载所有的模块，而不是只加载第一个匹配的模块后就停止，为的是让内核找到工作状况最良好的模块。当设备被删除时，这些脚本并不卸载任何模块。如果此时它们要卸载模块，由于控制着被删除设备的驱动程序可能还控制着其他设备，因而可能会导致其他设备的意外关闭。

请注意，现在 modprobe 程序可以直接从模块内读取 MODULE_DEVICE_TABLE，而不需要使用模块映像文件，热插拔脚本可以认为是对 modprobe 程序的简单包装。

udev

在内核中建立统一的驱动程序模型的一个主要原因是，允许用户空间用动态的方法管理*/dev*目录。通过在用户空间中实现devfs，这个目的已经达到，但是由于缺乏积极的维护和一些不可修改的bug，它的代码基础已经逐渐衰弱。许多内核开发者意识到，如果要把所有的设备信息输出到用户空间，就要对整个*/dev*进行必要的管理。

在devfs的设计中，有一些致命的缺陷。它需要修改每个设备的驱动程序以支持它，并且还需要设备驱动程序指定*/dev*目录树中的名称和位置。它还不能正确处理动态的主设备号和次设备号，另外它还不允许用户空间使用简单的方法改写设备名字，从而强制要求设备命名规则保存在内核中，而不是用户空间中。Linux内核开发者对在内核中保存规则深恶痛绝，并且由于devfs的命名规则不符合Linux Standard Base规范，使得他们非常烦恼。

随着Linux内核被安装在大型服务器上，许多用户都遇到了如何管理大量设备的问题。拥有10 000个独立设备的磁盘阵列提出了非常难解决的问题：如何确保特定的磁盘永远保持特定的名字，而与被放在磁盘阵列中的位置和被内核发现的时间无关。同样的问题也困扰着桌面系统的用户，当他们将两个USB打印机安装到系统上后，才发现无法保证重新启动系统后原来分配给一台打印机的*/dev/lp0*不被分配给另一台。

因此*udev*出现了。它依赖于sysfs输出到用户空间的所有设备信息，以及当设备添加或者删时，*/sbin/hotplug*对它的通知。比如为给定的设备命名等一些决策方法，可以在内核空间以外的用户空间指定。这保证了能从内核中删除命名规则，并且允许在为每个设备命名时具有很大的灵活性。

关于如何使用和配置*udev*，请参看发行版中*udev*软件包内的文档。

为了让*udev*能够正常工作，一个设备驱动程序要做的所有事情是：通过sysfs将驱动程序所控制设备的主设备号和次设备号导出到用户空间。对于那些使用子系统分配主设备号和次设备号的驱动程序，该工作已经由子系统完成，驱动程序不用做任何事。这样的子系统例子有tty、misc、usb、input、scsi、block、i2c、network和frame buffer子系统。如果驱动程序通过调用*cdev_init*函数或者是老版本的*register_chrdev*函数，自己处理获得的主设备号和次设备号，那么为了能正确使用*udev*，需要对驱动程序进行修改。

*udev*在sysfs中的*/class/*目录树中搜索名为dev的文件，这样内核通过*/sbin/hotplug*接口调用它的时候，就能获得分配给特定设备的主设备号和次设备号。一个设备驱动程序只需要为它所控制的每个设备创建该文件。class_simple接口是完成这件工作的最简单方法。

如同在“class_simple接口”一节中提到的，使用class_simple接口的第一步是通过*class_simple_create*函数创建class_simple结构：

```
static struct class_simple *foo_class;
...
foo_class = class_simple_create(THIS_MODULE, "foo");
if (IS_ERR(foo_class)) {
    printk(KERN_ERR "Error creating foo class.\n");
    goto error;
}
```

这段代码在sysfs中的*/sys/class/foo*下创建一个目录。

当驱动程序发现一个设备时，并且已经像第三章中介绍的那样分配了一个次设备号，驱动程序将调用*class_simple_device_add*函数：

```
class_simple_device_add(foo_class, MKDEV(FOO_MAJOR, minor), NULL, "foo%d", minor);
```

这段代码在*/sys/class/foo*下创建一个子目录*fooN*，这里*N*是设备的次设备号。在这个目录中创建一个文件——*dev*，有了这个文件，*udev*就可以为设备创建一个设备节点。

当设备与驱动程序脱离时，它也与分配的次设备号脱离，此时需要调用*class_simple_device_remove*函数删除该设备在sysfs中的入口项：

```
class_simple_device_remove(MKDEV(FOO_MAJOR, minor));
```

然后，当驱动程序完全关闭时，需要调用*class_simple_destroy*函数删除先前由*class_simple*函数创建的类：

```
class_simple_destroy(foo_class);
```

由*class_simple_device_add*函数创建的*dev*文件包含了主设备号和次设备号，它们被一个“:”分开。如果要在子系统class目录中提供其他文件，驱动程序将不使用class_simple接口，而是使用*print_dev_t*函数为指定的设备正确格式化主设备号和次设备号。

处理固件

作为一个驱动程序作者，可能会发现对于某些设备，必须先把固件下载到设备后，它才能正常工作。在硬件市场上的竞争非常激烈，制造商不愿为设备控制固件多用任何一点EEPROM。因此固件一般在随硬件发行的CD上，操作系统负责将固件传递到设备上。

使用类似下面的声明来处理固件问题：

```
static char my_firmware[] = { 0x34, 0x78, 0xa4, ... };
```

但是这个解决方法几乎是一个错误。将固件代码放入驱动程序会使驱动程序代码膨胀，使得固件升级困难，并容易导致许可证问题。供货商发布遵循GPL的固件映像文件是非常不可靠的，将其与GPL代码混合起来通常是个错误。因此不要把包含固件的驱动程序放入内核，或者包含在Linux发行版中。

内核固件接口

正确的做法是当需要的时候，从用户空间获得固件。一定不要从内核空间直接打开一个包含固件的文件。这是一个隐含错误的操作，因为它把策略（以文件名的形式）包含进了内核。相反，正确的方法是使用固件接口，这些接口就是为了这个目的而引入的：

```
#include <linux/firmware.h>
int request_firmware(const struct firmware **fw, char *name,
                     struct device *device);
```

*request_firmware*调用要求用户空间为内核定位并提供一个固件映像文件；我们一会儿将讲解它的工作细节。name表示需要的固件，通常name是供应商提供的固件文件名称。典型的文件名如*my_firmware.bin*。如果固件被正确加载，返回值是0（否则将返回错误代码），fw参数指向一个下面的结构：

```
struct firmware {
        size_t size;
        u8 *data;
};
```

该结构包含了实际的固件，现在可以把它下载到设备上了。请注意这个固件在用户空间内并未加以检验，因此，在将正确的固件映像传递给硬件前，请对其做部分或者全部的检测。设备固件通常包含校验字符串，比如校验和，我们首先对它们进行检验，然后才能信任这些数据。

在把固件发送到设备后，需要使用下面的函数释放内核中的结构：

```
void release_firmware(struct firmware *fw);
```

由于*request_firmware*需要用户空间的操作，因此在返回前它将保持睡眠状态。如果当驱动程序必须要使用固件，而又不能进入睡眠状态时，可以使用下面的异步函数：

```
int request_firmware_nowait(struct module *module,
                            char *name, struct device *device, void *context,
                            void (*cont)(const struct firmware *fw, void *context));
```

这里的附加参数是module（通常该参数是THIS_MODULE）、context（并不是固件

子系统使用的私有数据指针）和cont。如果一切正常，*request_firmware_nowait*将开始固件加载过程并返回0。过一段时间后，将使用加载的结果作为参数调用cont。如果由于某些原因固件加载失败，则fw是NULL。

工作原理

固件子系统使用sysfs和热插拔机制工作。当调用*request_firmware*时，在*/sys/class/firmware*下将创建一个目录，该目录使用设备名作为它的目录名。该目录包含三个属性：

loading
: 该属性由负责装载固件的用户空间进程设置为1。当装载过程完毕时，它将被设置为0。将loading设置为-1，将终止固件装载过程。

data
: data是一个二进制属性，用来接收固件数据。在设置完loading后，用户空间进程将把固件写入该属性。

device
: 该属性是到*/sys/devices*下相应入口的符号链接。

一旦sysfs入口被创建，内核将为设备产生热插拔事件。传递给热插拔处理程序的环境包括一个FIRMWARE变量，它将设置为提供给*request_firmware*的名字。处理程序定位固件文件，使用所提供的属性把固件文件拷贝到内核。如果不能发现固件文件，处理程序将设置*loading*属性为-1。

如果在10秒钟之内不能为固件的请求提供服务，内核将放弃努力并向驱动程序返回错误状态。这个超时值可以通过修改sysfs属性*/sys/class/firmware/timeout*来改变。

*request_firmware*接口允许使用驱动程序来发布设备的固件。当正确地整合进热插拔机制后，固件加载子系统允许设备不受干扰地工作。很明显这是处理该问题的最好方法。

然而还有一点要特别注意：不能在没有制造商许可的情况下发行设备的固件。一些制造商同意在某些条款保护下授权许可使用他们的固件，而一些制造商就不是那么配合了。无论哪种情况，没有他们的许可就拷贝和发行他们的固件，是违反版权法的，这可能会引起麻烦。

快速索引

在本章中已经介绍了许多函数，这里是对它们的一个总结。

kobject

```
#include <linux/kobject.h>
```

包含文件中包含了对kobject的定义，以及相关的结构和函数。

```
void kobject_init(struct kobject *kobj);
int kobject_set_name(struct kobject *kobj, const char *format, ...);
```

kobject的初始化函数。

```
struct kobject *kobject_get(struct kobject *kobj);
void kobject_put(struct kobject *kobj);
```

管理kobject引用计数的函数。

```
struct kobj_type;
struct kobj_type *get_ktype(struct kobject *kobj);
```

对包含kobject的结构类型的描述，使用*get_ktype*获得与指定kobject相关的kobj_type。

```
int kobject_add(struct kobject *kobj);
extern int kobject_register(struct kobject *kobj);
void kobject_del(struct kobject *kobj);
void kobject_unregister(struct kobject *kobj);
```

*kobject_add*向系统添加kobject，处理kset成员关系，sysfs表述以及产生热插拔事件。*kobject_register*函数是*kobject_init*和*kobject_add*的组合。使用*kobject_del*删除一个kobject，或者使用*kobject_unregister*函数，它是*kobject_del*和*kobject_put*的组合。

```
void kset_init(struct kset *kset);
int kset_add(struct kset *kset);
int kset_register(struct kset *kset);
void kset_unregister(struct kset *kset);
```

kset的初始化和注册函数。

```
decl_subsys(name, type, hotplug_ops);
```

使声明子系统得以简化的宏。

```
void subsystem_init(struct subsystem *subsys);
int subsystem_register(struct subsystem *subsys);
void subsystem_unregister(struct subsystem *subsys);
struct subsystem *subsys_get(struct subsystem *subsys)
void subsys_put(struct subsystem *subsys);
```

对子系统的操作。

sysfs 操作

```
#include <linux/sysfs.h>
```

包含 sysfs 声明的包含文件。

```
int sysfs_create_file(struct kobject *kobj, struct attribute *attr);
int sysfs_remove_file(struct kobject *kobj, struct attribute *attr);
int sysfs_create_bin_file(struct kobject *kobj, struct bin_attribute *attr);
int sysfs_remove_bin_file(struct kobject *kobj, struct bin_attribute *attr);
int sysfs_create_link(struct kobject *kobj, struct kobject *target, char
                      *name);
void sysfs_remove_link(struct kobject *kobj, char *name);
```

添加或删除与 kobject 相关属性文件的函数。

总线、设备和驱动程序

```
int bus_register(struct bus_type *bus);
void bus_unregister(struct bus_type *bus);
```

在设备模型中实现总线注册和注销的函数。

```
int bus_for_each_dev(struct bus_type *bus, struct device *start, void *data,
                     int (*fn)(struct device *, void *));
int bus_for_each_drv(struct bus_type *bus, struct device_driver *start, void
                    *data, int (*fn)(struct device_driver *, void *));
```

这些函数分别遍历附属于指定总线的每个设备和驱动程序。

```
BUS_ATTR(name, mode, show, store);
int bus_create_file(struct bus_type *bus, struct bus_attribute *attr);
void bus_remove_file(struct bus_type *bus, struct bus_attribute *attr);
```

使用宏 *BUS_ATTR* 声明了一个 bus_attribute 结构，使用上面的两个函数可对该结构进行添加和删除。

```
int device_register(struct device *dev);
void device_unregister(struct device *dev);
```

处理设备注册的函数。

```
DEVICE_ATTR(name, mode, show, store);
int device_create_file(struct device *device, struct device_attribute *entry);
void device_remove_file(struct device *dev, struct device_attribute *attr);
```

处理设备属性的宏和函数。

```
int driver_register(struct device_driver *drv);
void driver_unregister(struct device_driver *drv);
```

注册和注销设备驱动程序的函数。

```
DRIVER_ATTR(name, mode, show, store);
int driver_create_file(struct device_driver *drv, struct driver_attribute
                       *attr);
void driver_remove_file(struct device_driver *drv, struct driver_attribute
                        *attr);
```

管理驱动程序属性的宏和函数。

类

```
struct class_simple *class_simple_create(struct module *owner, char *name);
void class_simple_destroy(struct class_simple *cs);
struct class_device *class_simple_device_add(struct class_simple *cs, dev_t
  devnum, struct device *device, const char *fmt, ...);
void class_simple_device_remove(dev_t dev);
int class_simple_set_hotplug(struct class_simple *cs, int (*hotplug)(struct
  class_device *dev, char **envp, int num_envp, char *buffer, int
  buffer_size));
```

实现class_simple接口的函数；它们管理了包含dev属性和其他内容在内的简单类入口。

```
int class_register(struct class *cls);
void class_unregister(struct class *cls);
```

注册和注销类。

```
CLASS_ATTR(name, mode, show, store);
int class_create_file(struct class *cls, const struct class_attribute *attr);
void class_remove_file(struct class *cls, const struct class_attribute *attr);
```

处理类属性的常用宏和函数。

```
int class_device_register(struct class_device *cd);
void class_device_unregister(struct class_device *cd);
int class_device_rename(struct class_device *cd, char *new_name);
CLASS_DEVICE_ATTR(name, mode, show, store);
```

```
int class_device_create_file(struct class_device *cls, const struct
                             class_device_attribute *attr);
void class_device_remove_file(struct class_device *cls, const struct
                              class_device_attribute *attr);
```

实现类设备接口的函数和宏。

```
int class_interface_register(struct class_interface *intf);
void class_interface_unregister(struct class_interface *intf);
```

向类添加（或者删除）接口的函数。

固件

```
#include <linux/firmware.h>
int request_firmware(const struct firmware **fw, char *name, struct device
                     *device);
int request_firmware_nowait(struct module *module, char *name, struct device
  *device, void *context, void (*cont)(const struct firmware *fw, void
  *context));
void release_firmware(struct firmware *fw);
```

内核中实现固件加载的接口函数。

第十五章

内存映射和 DMA



本章将探讨Linux内存管理，并将重点放在对设备驱动程序编写者有价值的技术上。许多类型的驱动程序编程都需要了解一些虚拟内存子系统如何工作的知识；当遇到更为复杂、性能要求更为苛刻的子系统时，本章所讨论的内容迟早都要用到。虚拟内存子系统也是Linux内核中非常有趣的一部分，因此应当在这里花上一些时间。

本章的内容分为三个部分：

- 第一部分讲述了*mmap*系统调用的实现过程。该系统调用直接将设备内存映射到用户进程的地址空间里。虽然不是所有设备都需要*mmap*的支持，但是对一些设备来说，设备内存映射能显著地提高性能。
- 然后从另外一个角度，讲述如何跨越边界直接访问用户空间的内存页。一些相关的驱动程序需要这种能力；在许多情况下，内核执行了该种映射，而无需驱动程序的参与。但是了解用户空间内存如何映射到内核中的方法（使用get_user_pages）对我们很有帮助。
- 最后讲述了直接内存访问（DMA）I/O操作，它使得外设具有直接访问系统内存的能力。

当然，要掌握上述的技术，都需要了解Linux内存管理的工作原理，所以现在就从该子系统的概述讲起。

Linux的内存管理

本节将要关注Linux内存管理实现的主要特性，而非讲述操作系统中内存管理的理论。虽然，我们无需成为Linux虚拟内存的专家才能去实现*mmap*操作，但是了解它的工作概况还是大有益处的。下面将要用相当长的篇幅讲述内核用来管理内存的数据结构。当了解了必需的背景知识后，才能灵活运用这些结构。

地址类型

Linux 是一个虚拟内存系统，这意味着用户程序所使用的地址与硬件使用的物理地址是不等同的。虚拟内存引入了一个间接层，它使得许多操作成为可能。有了虚拟内存，在系统中运行的程序可以分配比物理内存更多的内存；甚至一个单独的进程都能拥有比系统物理内存更多的虚拟地址空间。虚拟地址还能让程序在进程的地址空间内使用更多的技巧，包括将程序的内存映射到设备内存上。

到目前为止，已经讨论了虚拟和物理地址，但是忽略了大量的细节。Linux 系统处理多种类型的地址，而每种类型的地址都有自己的语义。但不幸的是，在何种情况下使用何种类型的地址，内核代码并未明确加以区分，因此程序对此要仔细处理。

下面是一个Linux使用的地址类型列表。图15-1说明了这些地址类型与物理内存之间的关系。

用户虚拟地址

这是在用户空间程序所能看到的常规地址。用户地址或者是32位的，或者是64位的，这取决于硬件的体系架构。每个进程都有自己的虚拟地址空间。

物理地址

该地址在处理器和系统内存之间使用。物理地址也是32位或者64位长的，在某些情况下甚至32位系统也能使用64位的物理内存。

总线地址

该地址在外围总线和内存之间使用。通常它们与处理器使用的物理地址相同，但这么做并不是必需的。一些计算机体系架构提供了I/O内存管理单元（IOMMU），它实现总线和主内存之间的重新映射。IOMMU 可以用很多种方式让事情变得简单（比如使内存中的分散缓冲区对设备来说是连续的），但是当设置DMA操作时，编写IOMMU相关的代码是一个必需的额外步骤。当然总线地址是与体系架构密切相关的。

内核逻辑地址

内核逻辑地址组成了内核的常规地址空间。该地址映射了部分（或者全部）内存，并经常被视为物理地址。在大多数体系架构中，逻辑地址和与其相关联的物理地址的不同，仅仅是在它们之间存在一个固定的偏移量。逻辑地址使用硬件内建的指针大小，因此在安装了大量内存的32位系统中，它无法寻址全部的物理地址。逻辑地址通常保存在 `unsigned long` 或者 `void *` 这样类型的变量中。*kmalloc* 返回的内存就是内核逻辑地址。

内核虚拟地址

内核虚拟地址和逻辑地址的相同之处在于，它们都将内核空间的地址映射到物理地址上。内核虚拟地址与物理地址的映射不必是线性的和一对一的，而这是逻辑地址空间的特点。所有的逻辑地址都是内核虚拟地址，但是许多内核虚拟地址不是逻辑地址。举个例子，*vmalloc* 分配的内存具有一个虚拟地址（但并不存在直接的物理映射）。*kmap* 函数（在本章后面论述）也返回一个虚拟地址。虚拟地址通常保存在指针变量中。

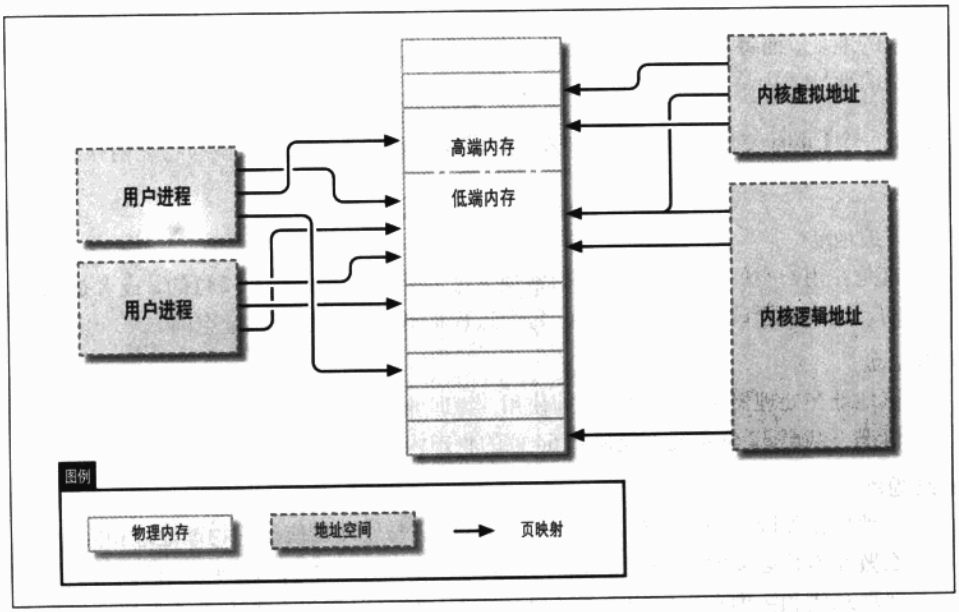


图 15-1：Linux 中使用的地址类型

如果有一个逻辑地址，宏 *__pa()*（在 *<asm/page.h>* 中定义）返回其对应的物理地址；使用宏 *__va()* 也能将物理地址逆向映射到逻辑地址，但这只对低端内存页有效。

不同的内核函数需要不同类型的地址。如果在C中已经定义好了不同的类型，那么需要的地址类型将很明确，但现实不是这样的。在本章中，将明确表述在何处使用何种类型的地址。

物理地址和页

物理地址被分成离散的单元，称之为页。系统内部许多对内存的操作都是基于单个页的。

每个页的大小随体系架构的不同而不同，但是目前大多数系统都使用每页4096个字节。常量 PAGE_SIZE（在 *<asm/page.h>* 中定义）给出了在任何指定体系架构下的页大小。

仔细观察内存地址，无论是虚拟的还是物理的，它们都被分为页号和一个页内的偏移量。举个例子，如果使用每页4096个字节，那么最后的12位是偏移量，而剩余的高位则指定了页号。如果忽略了地址偏移量，并将除去偏移量的剩余位移到右端，称该结果为页帧数。移动位以在页帧数和地址间进行转换是一个常用操作；宏 PAGE_SHIFT 将告诉程序员，必须移动多少位才能完成这个转换。

高端与低端内存

在装有大量内存的32位系统中，逻辑和内核虚拟地址的不同将非常突出。使用32位只能在4GB的内存中寻址。由于这种建立虚拟地址空间的问题，直到最近，32位系统的Linux仍被限制使用少于4GB的内存。

内核（在x86体系架构中，这是默认的设置）将4GB的虚拟地址空间分割为用户空间和内核空间；在二者的上下文中使用同样的映射。一个典型的分割是将3GB分配给用户空间，1GB分配给内核空间（注1）。内核代码和数据结构必须与这样的空间相匹配，但是占用内核地址空间最大的部分是物理内存的虚拟映射。内核无法直接操作没有映射到内核地址空间的内存。换句话说，内核对任何内存的访问，都需要使用自己的虚拟地址。因此许多年来，由内核所能处理的最大物理内存数量，就是将映射至虚拟地址空间内核部分的大小，再减去内核代码自身所占用的空间。因此，基于x86的Linux系统所能使用的最大物理内存，会比1GB小一点。

为了应对商业压力，在不破坏32位应用程序和系统兼容性的情况下，为了能使用更多的内存，处理器制造厂家为他们的产品增添了“地址扩展”特性。其结果是在许多情况下，即使32位的处理器都可以在大于4GB的物理地址空间寻址。然而有多少内存可以直接映射到逻辑地址的限制依然存在。只有内存的低端部分（依赖与硬件和内核的设置，一般为1到2GB）拥有逻辑地址（注2）；剩余的部分（高端内存）是没有的。在访问特定的高端内存页前，内核必须建立明确的虚拟映射，使该页可在内核地址空间中被访问。因此，许多内核数据结构必须被放置在低端内存中；而高端内存更趋向于为用户空间进程页所保留。

注1：许多非x86的体系架构不需要这里描述的内核/用户空间分割即可有效工作，因此，这些体系架构在32位系统上就能获得4GB的内核地址空间。但是，这一小节描述的限制对安装有多于4GB内存的系统仍然适用。

注2：2.6内核通过一个补丁可在x86硬件上支持“4G/4G”模式，它可以让内核和用户虚拟地址空间变大，但会引入微小的性能代价。

术语“高端内存”可能对一些人来说理解起来比较困难，特别是在PC世界中，它还有着其他的含义。因此为了弄清这个问题，这里先对它进行定义：

低端内存

存在于内核空间上的逻辑地址内存。几乎所有现在读者遇到的系统，它全部的内存都是低端内存。

高端内存

是指那些不存在逻辑地址的内存，这是因为它们处于内核虚拟地址之上。

在i386系统中，虽然在内核配置的时候能够改变低端内存和高端内存的界限，但是通常将该界限设置为小于1GB。这个界限与早期PC中的640K限制没有任何关系，并且它的设置也与硬件无关。相反它是由内核设置的，把32位地址空间分割成内核空间与用户空间。

在后面的部分中，将指出使用高端内存的限制。

内存映射和页结构

由于历史的关系，内核使用逻辑地址来引用物理内存中的页。然而由于支持了高端内存，就暴露出一个明显的问题 —— 在高端内存中将无法使用逻辑地址。因此内核中处理内存的函数趋向使用指向page结构的指针（在*<linux/mm.h>*中定义）。该数据结构用来保存内核需要知道的所有物理内存信息；对系统中每个物理页，都有一个page结构相对应。下面介绍该结构中包含的几个成员：

`atomic_t count;`

对该页的访问计数。当计数值为0时，该页将返回给空闲链表。

`void *virtual;`

如果页面被映射，则指向页的内核虚拟地址；如果未被映射则为NULL。低端内存页总是被映射；而高端内存页通常不被映射。并不是在所有体系架构中都有该成员；只有在页的内核虚拟地址不容易被计算时，它才被编译。如果要访问该成员，正确的方法是使用下面讲述的*page_address*宏。

`unsigned long flags;`

描述页状态的一系列标志。其中，PG_locked表示内存中的页已经被锁住，而PG_reserved表示禁止内存管理系统访问该页。

在page结构中还包含了许多信息，但这是深层次内存管理所关心的问题，而驱动程序作者不必要了解。

内核维护了一个或者多个page结构数组，用来跟踪系统中的物理内存。在一些系统中，有一个单独的数组称之为mem_map。在另外一些系统中，情况将会复杂很多。非一致性内存访问（Nonuniform Memory Access，NUMA）系统和有大量不连续物理内存的系统会有多个内存映射数组，因此从可移植性考虑，代码不要直接访问那些数组。幸运的是，通常只需要使用page结构的指针，而不需要了解它们是怎么来的。

有一些函数和宏用来在page结构指针与虚拟地址之间进行转换：

```
struct page *virt_to_page(void *kaddr);
```

该宏在*<asm/page.h>*中定义，负责将内核逻辑地址转换为相应的page结构指针。由于它需要一个逻辑地址，因此它不能操作*vmalloc*生成的地址以及高端内存。

```
struct page *pfn_to_page(int pfn);
```

针对给定的页帧号，返回page结构指针。如果需要的话，在将页帧号传递给*pfn_to_page*前，使用*pfn_valid*检查页帧号的合法性。

```
void *page_address(struct page *page);
```

如果地址存在的话，则返回页的内核虚拟地址。对于高端内存来说，只有当内存页被映射后该地址才存在。该函数定义在*<linux/mm.h>*中。在大多数情况下，要使用*kmap*而不是*page_address*。

```
#include <linux/highmem.h>
void *kmap(struct page *page);
void kunmap(struct page *page);
```

*kmap*为系统中的页返回内核虚拟地址。对于低端内存页来说，它只返回页的逻辑地址；对于高端内存，*kmap*在专用的内核地址空间创建特殊的映射。由*kmap*创建的映射需要用*kunmap*释放；对该种映射的数量是有限的，因此不要持有映射过长的时间。*kmap*调用维护了一个计数器，因此如果两个或是多个函数对同一页调用*kmap*，操作也是正常的。请注意当没有映射的时候，*kmap*将会休眠。

```
#include <linux/highmem.h>
#include <asm/kmap_types.h>
void *kmap_atomic(struct page *page, enum km_type type);
void kunmap_atomic(void *addr, enum km_type type);
```

*kmap_atomic*是*kmap*的高性能版本。每个体系架构都为原子的*kmap*维护着一个槽（专用页表入口）的列表；*kmap_atomic*的调用者必须告诉系统，type参数使用的是哪个槽。对驱动程序有意义的槽只有KM_USER0和KM_USER1（针对在用户空间中直接运行的代码），KM_IRQ0和KM_IRQ1（针对中断处理程序）。要注意的是原子的kmap必须原子地处理，也就是说，在拥有它的时候，代码不能进入睡眠状

态。还要注意的是在内核中，没有任何机制能防止这两个函数使用相同的槽，以及防止它们之间的相互干涉（虽然对每个CPU都有一套特定的槽）。在实际情况中，对原子的kmap槽的争夺并不会引起什么问题。

在研究例子代码时，以及在本章及后面的章节中，读者会看到如何使用这些函数。

页表

在任何现代的系统中，处理器必须使用某种机制，将虚拟地址转换为相应的物理地址。这种机制被称为页表；它基本上是一个多层树形结构，结构化的数组中包含了虚拟地址到物理地址的映射和相关的标志位。即使在不直接使用这种页表的体系架构中，Linux内核也维护了一系列的页表。

设备驱动程序执行了大量操作，用来处理页表。幸运的是，对驱动程序作者来说，在2.6版内核中删除了对页表直接操作的需求。因此这里不对它们做详细讲解；富有好奇心的读者可以阅读Daniel P. Bovet和Marco Cesati编写的《Understanding The Linux Kernel》（O'Reilly）一书了解详细情况。

虚拟内存区

虚拟内存区（VMA）用于管理进程地址空间中不同区域的内核数据结构。一个VMA表示在进程的虚拟内存中的一个同类区域：拥有同样权限标志位和被同样对象（一个文件或者交换空间）备份的一个连续的虚拟内存地址范围。它符合更宽泛的“段”的概念，但是把其描述成“拥有自身属性的内存对象”更为贴切。进程的内存映射（至少）包含下面这些区域：

- 程序的可执行代码（通常称为text）区域。
- 多个数据区，其中包含初始化数据（在开始执行的时候就拥有明确的值）、非初始化数据（BSS，注3）以及程序堆栈。
- 与每个活动的内存映射对应的区域。

查看*/proc/<pid/maps>*（其中的*pid*要替换为具体的进程ID）文件就能了解进程的内存区域。*/proc/self*是一个特殊的文件，因为它始终指向当前进程。下面是多个内存映射的例子（注释以斜体的方式给出）：

注3：*BSS这一名称是一个历史遗物，来自一条老的汇编操作称为“block started by symbol（符号定义的块）”。可执行文件的BSS段并不会存储在磁盘上，而是由内核将零页映射到BSS地址范围。*

```
# cat /proc/1/maps   look at init
08048000-0804e000 r-xp 00000000 03:01 64652      /sbin/init  text
0804e000-0804f000 rw-p 00006000 03:01 64652      /sbin/init  data
0804f000-08053000 rwxp 00000000 00:00 0          zero-mapped BSS
40000000-40015000 r-xp 00000000 03:01 96278      /lib/ld-2.3.2.so  text
40015000-40016000 rw-p 00014000 03:01 96278      /lib/ld-2.3.2.so  data
40016000-40017000 rw-p 00000000 00:00 0          BSS for ld.so
42000000-4212e000 r-xp 00000000 03:01 80290      /lib/tls/libc-2.3.2.so  text
4212e000-42131000 rw-p 0012e000 03:01 80290      /lib/tls/libc-2.3.2.so  data
42131000-42133000 rw-p 00000000 00:00 0          BSS for libc
bffff000-c0000000 rwxp 00000000 00:00 0          Stack segment
ffffe000-fffff000 ---p 00000000 00:00 0          vsyscall page

# rsh wolf cat /proc/self/maps  #### x86-64 (trimmed)
00400000-00405000 r-xp 00000000 03:01 1596291     /bin/cat    text
00504000-00505000 rw-p 00004000 03:01 1596291     /bin/cat    data
00505000-00526000 rwxp 00505000 00:00 0                       bss
3252200000-3252214000 r-xp 00000000 03:01 1237890 /lib64/ld-2.3.3.so
3252300000-3252301000 r--p 00100000 03:01 1237890 /lib64/ld-2.3.3.so
3252301000-3252302000 rw-p 00101000 03:01 1237890 /lib64/ld-2.3.3.so
7fbfffe000-7fc0000000 rw-p 7fbfffe000 00:00 0                 stack segment
ffffffffff600000-ffffffffffe00000 ---p 00000000 00:00 0       vsyscall page
```

每行都是用下面的形式表示的:

```
start-end perm offset major:minor inode image
```

在*/proc/*/maps*中的每个成员（除映像名外）都与vm_area_struct结构中的一个成员相对应:

start

end

该内存区域的起始处和结束处的虚拟地址。

perm

内存区域的读、写和执行权限的位掩码。该成员描述了允许什么样的进程能访问属于该区域的页。该成员的最后一个字母或者是p表示私有，或者是s表示共享。

offset

表示内存区域在映射文件中的起始位置。偏移量为0表示内存区域的起始位置映射到文件的开始位置。

major

minor

拥有映射文件的设备的主设备号和次设备号。对于设备映射来说，主设备号和次设备号指的是包含设备特殊文件的磁盘分区，该文件由用户而非设备自身打开。

inode

被映射的文件的索引节点号。

image

被映射文件（通常是一个可执行映像）的名称。

vm_area_struct 结构

当用户空间进程调用 *mmap*，将设备内存映射到它的地址空间时，系统通过创建一个表示该映射的新 VMA 作为响应。支持 *mmap* 的驱动程序（当然要实现 *mmap* 方法）需要帮助进程完成 VMA 的初始化。因此驱动程序作者为了能支持 *mmap*，需要对 VMA 有所了解。

现在来学习 vm_area_struct 结构（在 *<linux/mm.h>* 中定义）中最重要的成员。在设备驱动程序对 *mmap* 的实现中会使用到这些成员。请注意，为优化查找方法，内核维护了 VMA 的链表和树型结构，而 vm_area_struct 中的许多成员都是用来维护这个结构的。因此驱动程序不能任意创建 VMA，或者打破这种组织结构。VMA 的主要成员如下所示（请注意这些成员和刚才看到的 */proc* 文件输出之间的区别）：

```
unsigned long vm_start;
unsigned long vm_end;
```

该 VMA 所覆盖的虚拟地址范围。这是 */proc/*/maps* 中最前面的两个成员。

```
struct file *vm_file;
```

指向与该区域（如果存在的话）相关联的 file 结构指针。

```
unsigned long vm_pgoff;
```

以页为单位，文件中该区域的偏移量。当映射一个文件或者设备时，它是该区域中被映射的第一页在文件中的位置。

```
unsigned long vm_flags;
```

描述该区域的一套标志。驱动程序最感兴趣的标志是 VM_IO 和 VM_RESERVED。VM_IO 将 VMA 设置成一个内存映射 I/O 区域。VM_IO 会阻止系统将该区域包含在进程的核心转储中。VM_RESERVED 告诉内存管理系统不要将该 VMA 交换出去；大多数设备映射中都设置该标志。

```
struct vm_operations_struct *vm_ops;
```

内核能调用的一套函数，用来对该内存区进行操作。它的存在表示内存区域是一个内核“对象”，这点和在本书中使用的 file 结构很相似。

```
void *vm_private_data;
```

驱动程序用来保存自身信息的成员。

与vm_area_struct结构类似，vm_operations_struct结构也定义在*<linux/mm.h>*中，其中包含了下面列出的函数。这些操作只是用来处理进程的内存需求，并按照声明的顺序将它们列了出来。在本章后面的部分，将介绍如何实现其中的几个函数。

```
void (*open)(struct vm_area_struct *vma);
```

内核调用*open*函数，以允许实现VMA的子系统初始化该区域。当对VMA产生一个新的引用时（比如fork进程时），则调用这个函数。唯一的例外发生在*mmap*第一次创建VMA时；在这种情况下，需要调用驱动程序的*mmap*方法。

```
void (*close)(struct vm_area_struct *vma);
```

当销毁一个区域时，内核将调用*close*操作。请注意由于VMA没有使用相应的计数，所以每个使用区域的进程都只能打开和关闭它一次。

```
struct page *(*nopage)(struct vm_area_struct *vma, unsigned long address, int
                       *type);
```

当一个进程要访问属于合法VMA的页，但该页又不在内存中时，则为相关区域调用*nopage*函数（如果定义了的话）。在将物理页从辅助存储器中读入后，该函数返回指向物理页的page结构指针。如果在该区域没有定义*nopage*函数，则内核将为其分配一个空页。

```
int (*populate)(struct vm_area_struct *vm, unsigned long address, unsigned
  long len, pgprot_t prot, unsigned long pgoff, int nonblock);
```

在用户空间访问页前，该函数允许内核将这些页预先装入内存。一般来说，驱动程序不必实现*populate*方法。

内存映射处理

最后一个内存管理难题是处理内存映射结构，它负责整合所有其他的数据结构。在系统中的每个进程（除了内核空间的一些辅助线程外）都拥有一个struct mm_struct结构（在*<linux/sched.h>*中定义），其中包含了虚拟内存区域链表、页表以及其他大量内存管理信息，还包含一个信号灯（mmap_sem）和一个自旋锁（page_table_lock）。在task结构中能找到该结构的指针；在少数情况下当驱动程序需要访问它时，常用的办法是使用current->mm。请注意，多个进程可以共享内存管理结构，Linux就是用这种方法实现线程的。

为了能对Linux内存管理数据结构有一个通盘的了解，现在首先看看*mmap*系统调用是如何实现的。

mmap设备操作

在现代Unix系统中，内存映射是最吸引人的特征。对于驱动程序来说，内存映射可以提供给用户程序直接访问设备内存的能力。

使用*mmap*的一个例子是看一下X Window系统服务器的部分虚拟内存区域：

```
cat /proc/731/maps
000a0000-000c0000 rwxs 000a0000 03:01 282652      /dev/mem
000f0000-00100000 r-xs 000f0000 03:01 282652      /dev/mem
00400000-005c0000 r-xp 00000000 03:01 1366927     /usr/X11R6/bin/Xorg
006bf000-006f7000 rw-p 001bf000 03:01 1366927     /usr/X11R6/bin/Xorg
2a95828000-2a958a8000 rw-s fcc00000 03:01 282652  /dev/mem
2a958a8000-2a9d8a8000 rw-s e8000000 03:01 282652  /dev/mem
...
```

X服务器完整的VMA的清单很长，但这里对其中的大部分内容都不感兴趣。可以看到，有四个独立的*/dev/mem*的映射，它为我们揭示了X服务器如何使用显示卡工作的内幕。第一个映射开始位置是a0000，这是在640KB ISA结构中显示RAM的标准位置。往下可以看到更大的一块映射区域e8000000，其地址位于系统最大RAM地址之上。这是对显示适配器中显存的直接映射。

在*/proc/iomem*中也可以看到：

```
000a0000-000bffff : Video RAM area
000c0000-000cffff : Video ROM
000d1000-000d1fff : Adapter ROM
000f0000-000fffff : System ROM
d7f00000-f7efffff : PCI Bus #01
  e8000000-efffffff : 0000:01:00.0
fc700000-fcffffff : PCI Bus #01
  fcc00000-fcc0ffff : 0000:01:00.0
```

映射一个设备意味着将用户空间的一段内存与设备内存关联起来。无论何时当程序在分配的地址范围内读写时，实际上访问的就是设备。在X服务器例子中，使用*mmap*就能迅速而便捷地访问显卡内存。对于那些与此类似、性能要求苛刻的应用程序，直接访问能显著提高性能。

正如读者怀疑的那样，不是所有的设备都能进行*mmap*抽象的。比如像串口和其他面向流的设备就不能做这样的抽象。对*mmap*的另外一个限制是：必须以PAGE_SIZE为单位进行映射。内核只能在页表一级上对虚拟地址进行管理，因此那些被映射的区域必须

是PAGE_SIZE的整数倍，并且在物理内存中的起始地址也要求是PAGE_SIZE的整数倍。如果区域的大小不是页的整数倍，则内核强制指定比区域稍大一点的尺寸作为映射的粒度。

对于驱动程序来说这些限制并不是什么大问题，因为访问设备的程序都是与设备相关的。由于程序必须知道设备的工作过程，因此程序员不会被诸如页边界之类的需求所困扰。在某些非x86平台上工作的ISA设备面临更大的制约，因为它们的ISA硬件视图是不连续的。比如一些Alpha计算机视ISA内存为不可直接映射的8位、16位或者32位的离散项的集合。在这种情况下，根本无法使用*mmap*。无法将ISA地址直接映射到Alpha地址，是由于这两种系统间，存在着不兼容的数据传输规则。虽然早期的Alpha处理器只能解决32位和64位内存访问问题，但是对于ISA来说只能进行8位和16位的传输，没有办法透明地将一个协议映射到另外一个协议上。

当灵活使用*mmap*时，它具有很大的优势。比如在X服务器例子中，它负责和显存间读写大量数据；与使用*lseek/write*相比，将图形显示映射到用户空间极大地提高了吞吐量。另外一个典型例子是控制PCI设备的程序。大多数PCI外围设备将它们的控制寄存器映射到内存地址中，高性能的应用程序更愿意直接访问寄存器，而不是不停的调用*ioctl*去获得需要的信息。

*mmap*方法是file_operations结构的一部分，并且执行*mmap*系统调用时将调用该方法。使用*mmap*，内核在调用实际函数之前，就能完成大量的工作，因此该方法的原型与系统调用有着很大的不同。它也与诸如*ioctl*和*poll*不同，内核在调用那些函数前不用做什么工作。

系统调用有着以下声明（在*mmap(2)*手册页中描述）：

```
mmap (caddr_t addr, size_t len, int prot, int flags, int fd, off_t offset)
```

但是文件操作声明如下：

```
int (*mmap) (struct file *filp, struct vm_area_struct *vma);
```

该函数中的filp参数与第三章中介绍的一样，vma包含了用于访问设备的虚拟地址的信息。因此大量的工作由内核完成；为了执行*mmap*，驱动程序只需要为该地址范围建立合适的页表，并将vma->vm_ops替换为一系列的新操作就可以了。

有两种建立页表的方法：使用remap_pfn_range函数一次全部建立，或者通过*nopage* VMA方法每次建立一个页表。这两种方法有它各自的优势和局限性。这里首先介绍一次全部建立的方法，因为它最简单。从这开始，将会为实际的实现方法逐渐增加其复杂性：

使用 remap_pfn_range

remap_pfn_range 和 *io_remap_page_range* 负责为一段物理地址建立新的页表，它们有着如下的原型：

```
int remap_pfn_range(struct vm_area_struct *vma,
                    unsigned long virt_addr, unsigned long pfn,
                    unsigned long size, pgprot_t prot);
int io_remap_page_range(struct vm_area_struct *vma,
                        unsigned long virt_addr, unsigned long phys_addr,
                        unsigned long size, pgprot_t prot);
```

通常函数的返回值是0，或者是个负的错误码。现在来看看各参数的含义：

`vma`

虚拟内存区域，在一定范围内的页将被映射到该区域内。

`virt_addr`

重新映射时的起始用户虚拟地址。该函数为处于virt_addr和virt_addr+size之间的虚拟地址建立页表。

`pfn`

与物理内存对应的页帧号，虚拟内存将要被映射到该物理内存上。页帧号只是将物理地址右移PAGE_SHIFT位。在多数情况下，VMA结构中的vm_pgoff成员包含了用户需要的值。该函数对处于(pfn<<PAGE_SHIFT)到(pfn<<PAGE_SHIFT)+size之间的物理地址有效。

`size`

以字节为单位，被重新映射的区域大小。

`prot`

新VMA要求的“保护（protection）”属性。驱动程序能够（也应该）使用vma->vm_page_prot中的值。

remap_pfn_range 函数的参数非常简单，当调用 *mmap* 函数的时候，它们中大部分的值在VMA中提供。也许读者会奇怪，为什么会有两个函数呢？第一个函数（*remap_pfn_range*）是在pfn指向实际系统RAM的时候使用，而 *io_remap_page_range* 是在phys_addr指向I/O内存的时候使用。在实际应用中，除了SPARC外，对每个体系架构这两个函数是等价的，而在大多数情况下会使用 *remap_pfn_range* 函数。对于有可移植性要求的驱动程序，要使用与特定情形相符的 *remap_pfn_range* 变种。

另外复杂性也表现在缓存上：对设备内存的引用通常不能被处理器所缓存。系统的BIOS会正确设置缓存，但是也可以通过protection成员禁止缓存特定的VMA。不幸的是，在

这个层面上的禁止缓存是与处理器高度相关的。好奇的读者可以参考*drivers/char/mem.c*中的*pgprot_noncached*函数以了解其中细节。本书不对这个主题进行讨论。

一个简单的实现

如果驱动程序要将设备内存线性地映射到用户地址空间中，程序员基本上就只需要调用*remap_pfn_range*函数。下面的代码来自*drivers/char/mem.c*，并且揭示了在一个被称为*simple*（Simple Implementation Mapping Pages with Little Enthusiasm）的典型模块中，该任务是如何被完成的：

```
static int simple_remap_mmap(struct file *filp, struct vm_area_struct *vma)
{
    if (remap_pfn_range(vma, vma->vm_start, vm->vm_pgoff,
                vma->vm_end - vma->vm_start,
                vma->vm_page_prot))
        return -EAGAIN;

    vma->vm_ops = &simple_remap_vm_ops;
    simple_vma_open(vma);
    return 0;
}
```

可见，重新映射内存就是调用*remap_pfn_range*函数创建所需的页表。

为 VMA 添加操作

如上所述，vm_area_struct结构包含了一系列针对VMA的操作。现在来看看如何简单实现这些函数。本节提供了针对VMA的*open*和*close*操作。当进程打开或者关闭VMA时，会调用这些操作；特别是当fork进程或者创建一个新的对VMA引用时，随时都会调用*open*函数。对VMA函数*open*和*close*的调用由内核处理，因此它们没有必要重复内核中的工作。它们存在的意义在于为驱动程序处理其他所需要的事情。

除此之外，一个诸如*simple*这样的简单驱动程序不需再做什么特别的事情了。这里创建了*open*和*close*函数，它们负责向系统日志中输入信息，告诉系统它们被调用了。此外没有其他特殊用途了，不过它能告诉读者：如何提供这些函数以及何时调用它们。

因此，代码中用调用*printk*的新操作，覆盖了默认的vma->vm_ops：

```
void simple_vma_open(struct vm_area_struct *vma)
{
    printk(KERN_NOTICE "Simple VMA open, virt %lx, phys %lx\n",
            vma->vm_start, vma->vm_pgoff << PAGE_SHIFT);
}
```

```
void simple_vma_close(struct vm_area_struct *vma)
{
    printk(KERN_NOTICE "Simple VMA close.\n");
}

static struct vm_operations_struct simple_remap_vm_ops = {
    .open =  simple_vma_open,
    .close = simple_vma_close,
};
```

为了使这些操作对特定的映射有效，需要在相关VMA的vm_ops成员中保存指向simple_remap_vm_ops的指针。这通常在*mmap*方法中完成。如果回过头去看*simple_remap_mmap*示例，能看到如下代码：

```
vma->vm_ops = &simple_remap_vm_ops;
simple_vma_open(vma);
```

请注意对*simple_vma_open*函数的显式调用。由于在原来的*mmap*中没有调用*open*函数，因此必须显式调用它才能使其正常运行。

使用 nopage 映射内存

虽然*remap_page_range*在许多情况下工作良好，但并不能适应大多数的情况。有时驱动程序对*mmap*的实现必须具有更好的灵活性。在这种情形下，提倡使用VMA的*nopage*方法实现内存映射。

当应用程序要改变一个映射区域所绑定的地址时，会使用*mremap*系统调用，此时是使用*nopage*映射的最好的时机。当它发生时，内核并不直接告诉驱动程序什么时候*mremap*改变了映射VMA。如果VMA的尺寸变小了，内核将会刷新不必要的页，而不通知驱动程序。相反，如果VMA尺寸变大了，当调用*nopage*时为新页进行设置时，驱动程序最终会发现这个情况，因此没有必要做额外的通知工作。如果要支持*mremap*系统调用，就必须实现*nopage*函数。这里提供了设备中*nopage*的一个简单实现。

*nopage*函数具有以下原型：

```
struct page *(*nopage)(struct vm_area_struct *vma,
                       unsigned long address, int *type);
```

当用户要访问VMA中的页，而该页又不在内存中时，将调用相关的*nopage*函数。address参数包含了引起错误的虚拟地址，它已经被向下圆整到页的开始位置。*nopage*函数必须定位并返回指向用户所需要页的page结构指针。该函数还调用*get_page*宏，用来增加返回的内存页的使用计数：

```
get_page(struct page *pageptr);
```

该步骤对于保证映射页引用计数的正确性是非常必要的。内核为每个内存页都维护了该计数；当计数值为 0 时，内核将把该页放到空闲列表中。当 VMA 解除映射时，内核为区域内的每个内存页减小使用计数。如果驱动程序向区域添加内存页时不增加使用计数，则使用的计数值永远为 0，这将破坏系统的完整性。

Nopage 方法还能在 type 参数所指定的位置中保存错误的类型 —— 但是只有在 type 参数不为 NULL 的时候才行。在设备驱动程序中，type 的正确值应该总是 VM_FAULT_MINOR。

如果使用了 *nopage*，在调用 *mmap* 的时候，通常只需做一点点工作。示例代码如下：

```
static int simple_nopage_mmap(struct file *filp, struct vm_area_struct *vma)
{
    unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;

    if (offset >= __pa(high_memory) || (filp->f_flags & O_SYNC))
        vma->vm_flags |= VM_IO;
    vma->vm_flags |= VM_RESERVED;

    vma->vm_ops = &simple_nopage_vm_ops;
    simple_vma_open(vma);
    return 0;
}
```

mmap 函数的主要工作是将默认的 vm_ops 指针替换为自己的操作。然后 *nopage* 函数小心地每次“重新映射”一页，并且返回它的 page 结构指针。因为在这里实现了一个物理内存的窗口，重新映射的步骤非常简单：只是为需要的地址定位并返回了 page 结构的指针。*nopage* 函数的例子程序如下：

```
struct page *simple_vma_nopage(struct vm_area_struct *vma,
                unsigned long address, int *type)
{
    struct page *pageptr;
    unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
    unsigned long physaddr = address - vma->vm_start + offset;
    unsigned long pageframe = physaddr >> PAGE_SHIFT;

    if (!pfn_valid(pageframe))
        return NOPAGE_SIGBUS;
    pageptr = pfn_to_page(pageframe);
    get_page(pageptr);
    if (type)
        *type = VM_FAULT_MINOR;
    return pageptr;
}
```

再一次强调，这里只是简单映射了主内存，*nopage* 函数需要为失效地址查找正确的 page 结构，并且增加它的引用计数。因此所需要的步骤顺序是：首先计算物理地址，然后通

过右移PAGE_SHIFT位，将它转换成页帧号。由于用户空间能为用户提供它所拥有的任何地址，因此必须保证所用的页帧号合法；*pfn_valid*函数可以做这件事。如果地址超出了范围，将返回NOPAGE_SIGBUS，这会导致向调用进程发送一个总线信号。否则*pfn_to_page*函数获得所需要的page结构指针，这时，我们可以增加它的引用计数（使用*get_page*函数）并将其返回。

通常*nopage*方法返回一个指向page结构的指针。如果出于某些原因，不能返回一个正常的页（比如请求的地址超过了设备的内存区域），将返回NOPAGE_SIGBUS表示错误；这就是上面代码所做的事。*nopage*还能返回NOPAGE_OOM，表示由于资源紧张而造成的错误。

请注意，这个实现对ISA内存区域工作正常，但是不能在PCI总线上工作。PCI内存被映射到系统内存最高端之上，因此在系统内存映射中没有这些地址的入口。因为无法返回一个指向page结构的指针，所以*nopage*不能用于此种情形；在这种情况下，必须使用*remap_page_range*。

如果*nopage*函数是NULL，则负责处理页错误的内核代码将把零内存页映射到失效虚拟地址上。零内存页是一个写拷贝内存页，读它时会返回0，它被用于映射BSS段。任何一个引用零内存页的进程都会看到：一个充满了零的内存页。如果进程对内存页进行写操作，将最终修改私有拷贝。因此，如果一个进程调用*mremap*扩充一个映射区域，而驱动程序没有实现*nopage*，则进程将最终得到一块全是零的内存，而不会产生段故障错误。

重映射特定的I/O区域

迄今为止，所有例子都是对*/dev/mem*的再次实现；它们把物理内存重新映射到用户空间中。一个典型的驱动程序只映射与其外围设备相关的一小段地址，而不是映射全部地址。为了向用户空间只映射部分内存的需要，驱动程序只需要使用偏移量即可。下面的代码揭示了驱动程序如何对起始于物理地址simple_region_start（页对齐）、大小为simple_region_size字节的区域进行映射的工作过程：

```
unsigned long off = vma->vm_pgoff << PAGE_SHIFT;
unsigned long physical = simple_region_start + off;
unsigned long vsize = vma->vm_end - vma->vm_start;
unsigned long psize = simple_region_size - off;

if (vsize > psize)
    return -EINVAL; /* 跨度过大 */
remap_pfn_range(vma, vma_->vm_start, physical, vsize, vma->vm_page_prot);
```

当应用程序要映射比目标设备可用I/O区域大的内存时，除了计算偏移量，代码还检查参数的合法性并报告错误。在代码中，psize是偏移了指定距离后，剩下的物理I/O大小，vsize是虚拟内存需要的大小；该函数拒绝映射超出许可内存范围的地址。

请注意：用户进程总是使用*mremap*对映射进行扩展，有可能超过了物理内存区域的尾部。如果驱动程序没有定义一个*nopage*函数，它将不会获得这个扩展的通知，并且多出的区域将被映射到零内存页上。作为驱动程序作者，应该尽量避免这种情况的发生；将零内存页映射到区域的末端并非一件坏事，但是程序员也不愿意看到这种现象。

为防止扩展映射最简单的办法是实现一个简单的*nopage*方法，它会产生一个总线信号传递给故障进程。该函数有着类似于下面的形式：

```
struct page *simple_nopage(struct vm_area_struct *vma,
                           unsigned long address, int *type);
{ return NOPAGE_SIGBUS; /* 发送SIGBUS */}
```

如上所示，只有当进程抛弃那些存在于已知VMA中，但没有当前合法页表入口的地址时，才会调用*nopage*函数。如果使用*remap_pfn_range*映射全部的设备区域，将会为超过该区域的部分调用上面的*nopage*函数。因此它能安全返回NOPAGE_SIGBUS，通知错误的发生。当然一个更为彻底的*nopage*函数的实现会检查失效的地址是否在设备区域内，如果在设备区域内，它会执行重新映射。再强调一次，*nopage*函数不能对PCI内存进行操作，因此对PCI映射的扩展是不可能实现的。

重新映射RAM

对*remap_pfn_range*函数的一个限制是：它只能访问保留页和超出物理内存的物理地址。在Linux中，在内存映射时，物理地址页被标记为“保留的”（reserved），表示内存管理对其不起作用。比如在PC中，在640KB和1MB之间的内存被标记为保留的，因为这个范围位于内核自身代码的内部。保留页在内存中被锁住，并且是唯一可安全映射到用户空间的内存页；这个限制是保证系统稳定性的基本需求。

因此*remap_pfn_range*不允许重新映射常规地址，这包括调用*get_free_page*函数所获得的地址。相反它能映射零内存页。进程能访问私有的、零填充的内存页，而不是访问所期望的重新映射的RAM，除了这点外，一切工作正常。虽然如此，该函数还是做了大多数硬件设备驱动程序需要做的事，因为它能重新映射高端PCI缓冲区和ISA内存。

能够通过运行*mapper*看到对*remap_page_range*的限制，它是O'Reilly FTP服务器上*misc-progs*目录中的一个例子程序。*mapper*是一个用来快速检测*mmap*系统调用的易用工具；它根据命令行选项映射一个文件中的只读部分，并把映射区域的内容列在标准输

出上。比如在下面的会话中，显示了*/dev/mem*没有映射在64KB处的物理页，而是看到了一个全是零的内存页(运行该例子程序的主机是台PC，但在其他硬件平台上运行的结果应该一样)：

```
morgana.root# ./mapper /dev/mem 0x10000 0x1000 | od -Ax -t x1
mapped "/dev/mem" from 65536 to 69632
000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
001000
```

*remap_pfn_range*函数无法处理RAM表明：像*scull*这样基于内存的设备无法简单地实现*mmap*，因为它的设备内存是通用的RAM，而不是I/O内存。幸运的是，对任何需要将RAM映射到用户空间的驱动程序来说，有一种简单的方法可以达到目的，这就是前面介绍过的*nopage*函数。

使用 nopage 方法重映射 RAM

将实际的RAM映射到用户空间的方法是：使用vm_ops->nopage一次处理一个页错误。在第八章的*scullp*模块中，有一个实现该功能的例子。

*scullp*是一个面向内存页的字符设备。由于是面向内存页的，因此能对内存执行*mmap*。在执行内存映射的代码中，使用了在“Linux中的内存管理”一节中介绍的一些概念。

在学习代码前，先来看看影响*scullp*中*mmap*实现的一些设计选择：

- 只要映射了设备，*scullp*就不会释放设备内存。与其说是需求，还不如说是一种机制，这使得*scull*和其他类似设备有着很大的不同，因为打开它们进行写操作时，会把它们的长度截短为0。禁止释放一个被映射的*scullp*设备，使得一个进程改写被另外一个进程映射的区域成为可能，因此可以看到进程和设备内存是如何互动的。为了避免释放一个被映射的设备，驱动程序必须保存活动映射的计数；在device结构中的vmas成员的作用就是完成这一功能。
- 只有当*scullp*的order参数（在加载模块时设置）为0的时候，才执行内存映射。该参数控制了对*__get_free_pages*的调用（参看第八章中“get_free_page及相关函数”一节）。在*scullp*使用的分配函数——*__get_free_pages*函数内部实现体现了0幂次的限制（它强制每次只分配一个内存页，而不是一组）。为使分配性能最大化，Linux内核为每一个分配幂次维护了一个闲置页列表，而且只有簇中的第一个页的页计数可以由*get_free_pages*增加，并由*free_pages*减少。如果分配幂次大于0，则对*scullp*设备禁止使用*mmap*函数。因为*nopage*只处理单页而不处理一簇页面。*scullp*不知道如何为内存页正确管理引用计数，这是更高分配幂次的一部分

（如果需要复习一下 *scullp* 和内存分配幂次的值，可以返回到第八章的“使用一整页的 scull: scullp”一节）。

0 幂次的限制尽可能地简化了代码。通过处理页的使用计数，也有可能为多页分配正确地实现 *mmap*，但是这会增加例子的复杂性，却不能引入任何有趣的信息。

如果代码想要按照上面描述的规则来映射 RAM，就需要实现 *open*、*close* 和 *nopage* 等 VMA 方法，它也需要访问内存映像来调整页的使用计数。

scullp_mmap 的实现很是简短，因为它依赖 *nopage* 函数来完成所有的工作：

```
int scullp_mmap(struct file *filp, struct vm_area_struct *vma)
{
    struct inode *inode = filp->f_dentry->d_inode;

    /* 如果幂次不是 0，则禁止映射 */
    if (scullp_devices[iminor(inode)].order)
        return -ENODEV;

    /* 这里不做任何事情，“nopage”将填补这个空白 */
    vma->vm_ops = &scullp_vm_ops;
    vma->vm_flags |= VM_RESERVED;
    vma->vm_private_data = filp->private_data;
    scullp_vma_open(vma);
    return 0;
}
```

`if` 语句的目的是为了避免映射分配幂次不为 0 的设备。*scullp* 的操作被存储在 `vm_ops` 成员中，而且一个指向 device 结构的指针被存储在 `vm_private_data` 成员中。最后 `vm_ops->open` 被调用，以更新模块的使用计数和设备的活动映射计数。

open 和 *close* 函数只是简单地跟踪这些计数，其定义如下：

```
void scullp_vma_open(struct vm_area_struct *vma)
{
    struct scullp_dev *dev = vma->vm_private_data;

    dev->vmas++;
}

void scullp_vma_close(struct vm_area_struct *vma)
{
    struct scullp_dev *dev = vma->vm_private_data;

    dev->vmas--;
}
```

大部分工作由 *nopage* 函数完成。在 *scullp* 的实现中，*nopage* 的 `address` 参数用来计算设备里的偏移量，然后使用该偏移量在 *scullp* 的内存树中查找正确的页：

```
struct page *scullp_vma_nopage(struct vm_area_struct *vma,
                               unsigned long address, int *type)
{
    unsigned long offset;
    struct scullp_dev *ptr, *dev = vma->vm_private_data;
    struct page *page = NOPAGE_SIGBUS;
    void *pageptr = NULL; /* 默认值是"没有" */

    down(&dev->sem);
    offset = (address - vma->vm_start) + (vma->vm_pgoff << PAGE_SHIFT);
    if (offset >= dev->size) goto out; /* 超出范围 */

    /*
     * 现在从链表中获得了scullp设备以及内存页。
     * 如果设备有空白区，当进程访问这些空白区时，进程会收到SIGBUS。
     */

    offset >>= PAGE_SHIFT; /* offset是页号 */
    for (ptr = dev; ptr && offset >= dev->qset;) {
        ptr = ptr->next;
        offset -= dev->qset;
    }
    if (ptr && ptr->data) pageptr = ptr->data[offset];
    if (!pageptr) goto out; /* 空白区或者是文件末尾 */
    page = virt_to_page(pageptr);

    /* 获得该值，现在可以增加计数了 */
    get_page(page);
    if (type)
        *type = VM_FAULT_MINOR;
  out:
    up(&dev->sem);
    return page;
}
```

scullp 使用了由 *get_free_pages* 函数获得的内存。该内存使用逻辑地址寻址，因此 *scullp_nopage* 要做的全部工作就是调用 *virt_to_page* 来获得 page 结构的指针。

scullp 设备现在如预期的那样工作了，下面是 *mapper* 工具的输出。这里发送一个 */dev*（很长）目录清单给 *scullp* 设备，然后使用 *mapper* 工具查看 *mmap* 生成的清单片段：

```
morgana% ls -l /dev > /dev/scullp
morgana% ./mapper /dev/scullp 0 140
mapped "/dev/scullp" from 0 (0x00000000) to 140 (0x0000008c)
total 232
crw-------    1 root     root      10,  10 Sep 15 07:40 adbmouse
crw-r--r--    1 root     root      10, 175 Sep 15 07:40 agpgart
morgana% ./mapper /dev/scullp 8192 200
mapped "/dev/scullp" from 8192 (0x00002000) to 8392 (0x000020c8)
d0h1494
brw-rw----    1 root     floppy     2,  92 Sep 15 07:40 fd0h1660
brw-rw----    1 root     floppy     2,  20 Sep 15 07:40 fd0h360
brw-rw----    1 root     floppy     2,  12 Sep 15 07:40 fd0H360
```

重新映射内核虚拟地址

虽然很少需要重新映射内核虚拟地址，但是知道驱动程序是如何使用*mmap*将内核虚拟地址映射到用户空间的，也是一件有趣的事。一个真正的内核虚拟地址，就是诸如*vmalloc*这样的函数返回的地址——也就是说，是一个映射到内核页表的虚拟地址。本节中的代码是从*scullv*中抽取出来的，*scullv*是一个与*scullp*类似的模块，但它是通过*vmalloc*分配存储空间的。

除了不需要检查控制内存分配的`order`参数之外，*scullv*中的大多数实现与前面讨论的*scullp*中的一样。这是因为*vmalloc*每次只分配一个内存页，而单页分配比多页分配成功的可能性更高一些，因此分配的幂次问题在*vmalloc*所分配的空间中不存在。

除了上述部分，只有*scullp*和*scullv*所实现的*nopage*函数是不一样的。请记住*scullp*一旦发现了感兴趣的页，将调用*virt_to_page*获得相应的`page`结构指针。但是该函数不能在内核虚拟空间中使用，因此必须使用*vmalloc_to_page*替换它。*scullv*版本的*nopage*函数的最后部分如下：

```
    /*
     * 在scullv查找之后，"page"现在是当前进程所需要的页地址。
     * 由于它是一个vmalloc返回的地址，将其转化为一个page结构。
     */
    page = vmalloc_to_page(pageptr);

    /* 获得该值，现在增加它的计数 */
    get_page(page);
    if (type)
        *type = VM_FAULT_MINOR;
out:
    up(&dev->sem);
    return page;
```

出于对上述讨论内容的考虑，读者可能想要将*ioremap*返回的地址映射到用户空间上。但这么做是错误的，这是因为*ioremap*返回的地址比较特殊，不能把它当作普通的内核虚拟地址，应该使用*remap_pfn_range*函数将I/O内存重新映射到用户空间上。

执行直接I/O访问

内核缓冲了大多数I/O操作。对内核空间缓冲区的使用，在一定程度上分隔了用户空间和实际设备；这种分隔在许多情况下使得程序更容易实现，并且提高了性能。然而有些时候，直接对用户空间缓冲区执行I/O操作效果也是很好的。如果需要传输的数据量非常大，直接进行数据传输，而不需要额外地从内核空间拷贝数据操作的参与，这将会大大提高速度。

在2.6内核中一个使用直接I/O操作的例子是SCSI磁带机驱动程序。数据磁带会把大量数据传递给系统，而磁带的传输通常是面向记录的，因此在内核中缓冲数据的收益非常小。因此当条件成熟（比如用户空间缓冲区很大）的时候，SCSI磁带机驱动程序不通过数据拷贝，直接执行它的I/O操作。

然而必须要清醒的认识到，直接I/O并不能像人们期望的那样，总是能提供性能上的飞跃。设置直接I/O（这包括减少和约束相关的用户页）的开销非常巨大，而又没有使用缓存I/O的优势。比如，使用直接I/O需要*write*系统调用同步执行；否则应用程序将会不知道什么时候能再次使用它的I/O缓冲区。在每个写操作完成之前不能停止应用程序，这样会导致关闭程序缓慢，这就是为什么使用直接I/O的应用程序也使用异步I/O的原因。

无论如何，在字符设备中执行直接I/O是不可行的，也是有害的。只有确定设置缓冲I/O的开销特别巨大，才使用直接I/O。请注意块设备和网络设备根本不用担心实现直接I/O的问题；在这两种情况中，内核中高层代码设置和使用了直接I/O，而驱动程序级的代码甚至不需要知道已经执行了直接I/O。

在2.6内核中，实现直接I/O的关键是名为*get_user_pages*的函数，它定义在*<linux/mm.h>*中，并有以下原型：

```
int get_user_pages(struct task_struct *tsk,
                   struct mm_struct *mm,
                   unsigned long start,
                   int len,
                   int write,
                   int force,
                   struct page **pages,
                   struct vm_area_struct **vmas);
```

该函数有许多参数：

`tsk`

指向执行I/O的任务指针；它的主要目的是告诉内核，当设置缓冲区时，谁负责解决页错误的问题。该参数几乎总是`current`。

`mm`

指向描述被映射地址空间的内存管理结构的指针。`mm_struct`结构用来聚合进程虚拟地址空间中的VMA。对驱动程序来说，该参数总是`current->mm`。

`start`

`len`

`start`是用户空间缓冲区的地址（页对齐），`len`是页内的缓冲区长度。

write
force

如果write非零，对映射的页有写权限（意味着用户空间执行了读操作）。force标志告诉*get_user_pages*函数不要考虑对指定内存页的保护，直接提供所请求的访问；驱动程序对该参数总是设置为0。

pages
vmas

输出参数。如果调用成功，pages中包含了一个描述用户空间缓冲区page结构的指针列表，vmas包含了相应VMA的指针。显然这些参数指向的数组至少包含了len个指针。这两个参数都可以为NULL，但至少page结构指针要对缓冲区进行实际的操作。

*get_user_pages*函数是一个底层内存管理函数，使用了比较复杂的接口。它还需要在调用前，将*mmap*为获得地址空间的读取者/写入者信号量设置为读模式。因此，对*get_user_pages*的调用有类似以下的代码：

```
down_read(&current->mm->mmap_sem);
result = get_user_pages(current, current->mm, ...);
up_read(&current->mm->mmap_sem);
```

返回的值是实际被映射的页数，它可能会比请求的数量少（但是大于0）。

如果调用成功，调用者就会拥有一个指向用户空间缓冲区的页数组，它将被锁在内存中。为了能直接操作缓冲区，内核空间的代码必须用*kmap*或者*kmap_atomic*函数将每个page结构指针转换成内核虚拟地址。使用直接I/O的设备通常使用DMA操作，因此驱动程序要从page结构指针数组中创建一个分散/聚集链表。我们将在“分散/聚集映射”一节中对其进行详细讲述。

一旦直接I/O操作完成，就必须释放用户内存页。在释放前，如果改变了这些页中的内容，则必须通知内核，否则内核会认为这些页是“干净”的，也就是说，内核会认为它们与交换设备中的拷贝是匹配的，因此，无需回存就能释放它们。因此，如果改变了页（响应用户空间的读取请求），则必须使用下面的函数标记出每个被改变的页：

```
void SetPageDirty(struct page *page);
```

这个宏定义在*<linux/page-flags.h>*中。执行该操作的大多数代码首先要检查页，以确保该页不在内存映射的保留区内，因为这个区的页是不会被交换出去的，因此有如下代码：

```
if (! PageReserved(page))
    SetPageDirty(page);
```

由于用户空间内存通常不会被标记为保留，因此这个检查并不是严格要求的。但是，在对内存管理子系统有更深入的了解前，最好谨慎和细致些。

不管页是否被改变，它们都必须从页缓存中释放，否则它们会永远存在在那里。所需要使用的函数是：

```
void page_cache_release(struct page *page);
```

当然，如果需要的话，在页被标记为改变（dirty）后，应该执行该调用。

异步 I/O

添加到2.6内核中的一个新特性是异步I/O。异步I/O允许用户空间初始化操作，但不必等待它们完成，这样，当I/O在执行时，应用程序可以进行其他的操作。一个复杂的、高性能的应用程序也能使用异步I/O，让多个操作同时进行。

异步I/O的实现是可选的，只有少数驱动程序作者需要考虑这个问题，大多数设备并不能从异步操作中获得好处。在后面的几章中，块设备和网络设备驱动程序是完全异步操作的，因此只有字符设备驱动程序需要清楚地表示需要异步I/O的支持。如果有恰当的理由需要在同一时刻执行多于一个的I/O操作，则字符设备将会从异步I/O中受益。一个良好的例子是磁带机驱动程序，如果它的I/O操作不能以足够快的速度执行，则驱动器会显著变慢。一个为了获得该驱动器最优性能的应用程序应该使用异步I/O，同时准备执行多个操作。

针对于少数需要实现异步I/O的驱动程序作者，我们这里对异步I/O的工作过程做一个简要的介绍。在本章中讲述异步I/O的原因，是由于它的实现总是包含直接I/O操作（如果在内核中缓冲数据，则可以实现异步操作，而不给用户空间增加复杂程度）。

支持异步I/O的驱动程序应该包含<*linux/aio.h*>。有三个用于实现异步I/O的*file_operations*方法：

```
ssize_t (*aio_read) (struct kiocb *iocb, char *buffer,
                     size_t count, loff_t offset);
ssize_t (*aio_write) (struct kiocb *iocb, const char *buffer,
                      size_t count, loff_t offset);
int (*aio_fsync) (struct kiocb *iocb, int datasync);
```

*aio_fsync*操作只对文件系统有意义，因此不作深入讨论。另外两个函数，*aio_read*和*aio_write*与常用的*read*和*write*函数非常类似，但是也有一些不同。其中一个不同是：传递的`offset`参数是一个值；异步操作从不改变文件的位置，因此没有必要向它传递指针。这两个函数都使用`iocb`（I/O控制块，I/O control block）参数，一会将讨论它。

*aio_read*和*aio_write*函数的目的是初始化读和写操作，在这两个函数完成时，读写操作可能已经完成，也可能尚未完成。如果操作立刻完成，则函数将返回常规状态：传输的字节数或者是负的错误码。因此如果驱动程序作者自己的*read*函数称为*my_read*，下面的*aio_read*函数就是完全正确的（虽然是无意义的）：

```
static ssize_t my_aio_read(struct kiocb *iocb, char *buffer,
                           ssize_t count, loff_t offset)
{
    return my_read(iocb->ki_filp, buffer, count, &offset);
}
```

请注意file结构指针保存在kiocb结构中的ki_filp成员里。

如果支持异步I/O，则必须知道一个事实：内核有时会创建“同步IOCB”。也就是说异步操作实际上必须同步执行。读者也许会问：为什么会这样？但最好还是适应内核的要求。同步操作会在IOCB中标识，因此，驱动程序应该使用下面的函数进行查询：

```
int is_sync_kiocb(struct kiocb *iocb);
```

如果该函数返回非零值，则驱动程序必须执行同步操作。

最后的关键点是如何允许异步操作。如果驱动程序可以开始操作（或者简单点，将操作压入队列，等待未来某个时刻执行），它必须做两件事：记住与操作相关的所有信息，并且返回-EIOCBQUEUED给调用者。记住操作的信息包括了安排对用户空间缓冲区的访问；一旦返回，因为要运行在调用进程的上下文中，所以将不能再访问这个缓冲区。通常这意味着建立直接的内核映射（使用*get_user_pages*）或者DMA映射。-EIOCBQUEUED错误码表明操作还没有完成，它最终的状态将在未来某个时刻公布。

当未来某个时刻到来时，驱动程序必须通知内核操作已经完成。这需要使用*aio_complete*函数：

```
int aio_complete(struct kiocb *iocb, long res, long res2);
```

这里，iocb与最初传递给我们的IOCB相同，res是操作的结果状态，res2是返回给用户空间的第二状态码，大多数异步I/O会将res2设置为0。一旦调用了*aio_complete*，就不能再访问IOCB或者用户缓冲区了。

异步I/O例子

在例子源代码中，面向内存页的*scullp*驱动程序实现了异步I/O。该实现非常简单，但对于揭示异步操作是如何进行的，就已经足够了。

*aio_read*和*aio_write*函数实际上没做什么事：

```
static ssize_t scullp_aio_read(struct kiocb *iocb, char *buf, size_t count,
        loff_t pos)
{
    return scullp_defer_op(0, iocb, buf, count, pos);
}

static ssize_t scullp_aio_write(struct kiocb *iocb, const char *buf,
        size_t count, loff_t pos)
{
    return scullp_defer_op(1, iocb, (char *) buf, count, pos);
}
```

这些函数只是简单地调用了一个常用函数:

```
struct async_work {
    struct kiocb *iocb;
    int result;
    struct work_struct work;
};

static int scullp_defer_op(int write, struct kiocb *iocb, char *buf,
        size_t count, loff_t pos)
{
    struct async_work *stuff;
    int result;
    /* 虽然可以访问缓冲区，但现在要进行拷贝操作 */
    if (write)
        result = scullp_write(iocb->ki_filp, buf, count, &pos);
    else
        result = scullp_read(iocb->ki_filp, buf, count, &pos);

    /* 如果这是一个同步的IOCB，则现在返回状态值 */
    if (is_sync_kiocb(iocb))
        return result;

    /* 否则把完成操作向后推迟几毫秒 */
    stuff = kmalloc (sizeof (*stuff), GFP_KERNEL);
    if (stuff = = NULL)
        return result;
    /* 没有可用内存了，使之完成 */
    stuff->iocb = iocb;
    stuff->result = result;
    INIT_WORK(&stuff->work, scullp_do_deferred_op, stuff);
    schedule_delayed_work(&stuff->work, HZ/100);
    return -EIOCBQUEUED;
}
```

一个更完整的实现应该使用*get_user_pages*函数，以便将用户缓冲区映射到内核空间，为了简单起见，这里只是从起始位置拷贝了数据。然后调用*is_sync_kiocb*函数检查操作是否必须以同步方式完成。如果是，返回结果状态；如果不是，将相关信息保存在一个小结构中，然后安排作业队列，接着返回-EIOCBQUEUED。

到此为止，将控制权返回给了用户空间。

接着，作业队列执行了完整操作：

```
static void scullp_do_deferred_op(void *p)
{
    struct async_work *stuff = (struct async_work *) p;
    aio_complete(stuff->iocb, stuff->result, 0);
    kfree(stuff);
}
```

这里仅仅使用保存的信息调用了 *aio_complete* 函数。一个实际驱动程序的异步 I/O 实现当然比这复杂，但是其基本模式不会变。

直接内存访问

直接内存访问，或者 DMA，是关于内存问题讨论的高级部分。DMA 是一种硬件机制，它允许外围设备和主内存之间直接传输它们的 I/O 数据，而不需要系统处理器的参与。使用这种机制可以大大提高与设备通信的吞吐量，因为免除了大量的计算开销。

DMA 数据传输概览

在介绍编程细节之前，先回顾一下 DMA 传输是如何发生的，为了简化问题，只考虑输入传输。

有两种方式引发数据传输：或者是软件对数据的请求（比如通过 *read* 函数），或者是硬件异步地将数据传递给系统。

在第一种情况中，所需要的步骤概括如下：

1. 当进程调用 *read*，驱动程序函数分配一个 DMA 缓冲区，并让硬件将数据传输到这个缓冲区中。进程处于睡眠状态。
2. 硬件将数据写入到 DMA 缓冲区中，当写入完毕，产生一个中断。
3. 中断处理程序获得输入的数据，应答中断，并且唤醒进程，该进程现在即可读取数据。

第二种情况发生在异步使用 DMA 时。比如对于一个数据采集设备，即使没有进程读取数据，它也不断地写入数据。此时，驱动程序应该维护一个缓冲区，其后的 read 调用将返回所有积累的数据给用户空间。这种传输方式的步骤有所不同：

1. 硬件产生中断，宣告新数据的到来。

2. 中断处理程序分配一个缓冲区，并且告诉硬件向哪里传输数据。

3. 外围设备将数据写入缓冲区，完成后产生另外一个中断。

4. 处理程序分发新数据，唤醒任何相关进程，然后执行清理工作。

另一种异步方法可在网卡中看到。网卡期望在内存中建有一个循环缓冲区（通常叫做DMA环形缓冲区），并与处理器共享；每个输入的数据包都放入缓冲器环中的下一个可用缓冲器中，然后引发中断。接着驱动程序将数据包发送给内核其他部分处理，并在环形缓冲区中放置一个新的DMA缓冲区。

上述情况的处理步骤强调，高效的DMA处理依赖于中断报告。虽然可以使用轮询的驱动程序实现DMA，但这没有意义，因为一个轮询驱动程序会将DMA相对于简单的处理器驱动I/O获得的性能优势抵消掉（注4）。

这里介绍的另外一个相关术语是DMA缓冲区。DMA需要设备驱动程序分配一个或者多个适合执行DMA的特殊缓冲区。请注意许多驱动程序在初始化阶段分配了它们的缓冲区，并且一直使用它们直到关闭——在前面涉及到的“分配”一词含义是“保持一个已经分配的缓冲区”。

分配DMA缓冲区

本节主要讨论在低层分配DMA缓冲区的方法，很快就会介绍一个较高层的接口，但仍要正确理解这里介绍的内容。

使用DMA缓冲区的主要问题是：当大于一页时，它们必须占据连续的物理页，这是因为设备使用ISA或者PCI系统总线传输数据，而这两种方式使用的都是物理地址。有趣的是，这种限制对SBus（参看第十二章中“SBus”一节）是无效的，因为它在外围总线上使用了虚拟地址。某些体系架构的PCI总线也可以使用虚拟地址，但是出于可移植性的考虑，我们不建议使用这个特性。

虽然既可以在系统启动时，也可以在运行时分配DMA缓冲区，但是模块只能在运行时刻分配它们的缓冲区（在第八章中论述了该技术，其中的“获得更多缓冲区”一节讲述了在系统启动时分配的方法，而“kmalloc”和“get_free_page及其辅助函数”一节中描述了运行时分配的方法）。驱动程序作者必须谨慎地为DMA操作分配正确的内存类型，因为并不是所有内存区间都适合DMA操作。在实际操作中，一些设备和一些系统中的高端内存不能用于DMA，这是因为外围设备不能使用高端内存的地址。

注4：当然，任何事情都有例外。请阅读第十七章的“缓解接收中断”，其中说明了如何使用轮询实现最好的高性能网络驱动程序。

在现代总线上的大多数设备能够处理32位地址，这意味着常用的内存分配机制能很好地工作。一些PCI设备没能实现全部的PCI标准，因此不能使用32位地址，而一些ISA设备还局限在使用24位地址的阶段。

对于有这些限制的设备，应使用GFP_DMA标志调用*kmalloc*或者*get_free_pages*从DMA区间分配内存。当设置了该标志时，只有使用24位寻址方式的内存才能被分配。另外，还可以使用通用DMA层（不久会讲到）来分配缓冲区，这样也能满足对设备限制的需求。

DIY分配

读者已经知道*get_free_pages*函数可以分配多达几M字节的内存（最高可以达到MAX_ORDER，目前是11），但是对较大数量的请求，甚至是远少于128KB的请求也通常会失败，这是因为此时系统内存中充满了内存碎片（注5）。

当内核不能返回请求数量的内存，或者需要大于128KB内存（比如PCI帧捕获卡的普遍请求）的时候，相对于返回-ENOMEM，另外一个办法是在引导时分配内存，或者为缓冲区保留顶部物理RAM。我们已经在第八章的“获得更多缓冲区”一节讲述了如何在引导时分配内存，但对模块来说这是不可行的。在引导时，我们可以通过向内核传递“mem=参数”的办法保留顶部的RAM。比如系统有256MB内存，参数“mem=255M”将使内核不能使用顶部的1M字节。随后，模块可以使用下面的代码获得对该内存的访问权：

```
dmabuf = ioremap (0xFF00000 /* 255M */, 0x100000 /* 1M */);
```

随本书附带的例子代码中有一个分配器，它提供了一个API用来探测和管理保留的RAM，并且在多种体系架构中能成功使用。但是该分配器不能在配置有高端内存的系统上使用（比如物理内存数量超出CPU地址空间的系统）。

还有一个办法是使用GFP_NOFAIL分配标志来为缓冲区分配内存。但是该方法为内存管理子系统带来了相当大的压力，因此为整个系统带来了风险。所以，如果不是实在没有其他更好的方法，最好不要使用这个标志。

如果需要为DMA缓冲区分配一大块内存，最好考虑一下是否有替代的方法。如果设备支持分散/聚集I/O，则可以将缓冲区分配成多个小块，设备会很好地处理它们。当在用

注5：“碎片”一词通常用于磁盘，用来说明在磁介质上，文件并不是连续存放的。相同的概念也可以应用于内存，由于每个虚拟地址空间分散在整个物理RAM中，因此当请求DMA缓冲区时，也难以获得连续的空闲页。

户空间中执行直接I/O的时候，也可以用分散/聚集I/O。当需要有一大块缓冲区的时候，这是最好的解决方案。

总线地址

使用DMA的设备驱动程序将与连接到总线接口上的硬件通信，硬件使用的是物理地址，而程序代码使用的是虚拟地址。

实际上情况比这还要复杂些。基于DMA的硬件使用总线地址，而非物理地址。虽然ISA和PCI总线地址只是PC上的简单物理地址，但是对其他平台来说，却不总是这样。有时接口总线是通过将I/O地址映射到不同物理地址的桥接电路连接的。某些系统甚至有页面映射调度，能够使任意页面在外围总线上表现为连续的。

在最底层（一会将要介绍较高层的解决方案），Linux内核通过输出在*<asm/io.h>*中定义的一些函数，提供了可移植的方案。我们不推荐使用这些函数，因为只有在那些拥有非常简单I/O的体系架构中，它们才能工作正常；虽然如此，在阅读内核代码时还是会遇到它们。

```
unsigned long virt_to_bus(volatile void *address);
void *bus_to_virt(unsigned long address);
```

这些函数在内核逻辑地址和总线地址间执行了简单的转换。但对于必须使用I/O内存管理单元或者必须使用回弹缓冲区的情况下，它们将不能工作。执行这些转换的正确方法是使用通用DMA层，因此现在来讨论这个主题。

通用DMA层

DMA操作最终会分配缓冲区，并将总线地址传递给设备。一个可移植的驱动程序要求对所有体系架构都能安全而正确地执行DMA操作，编写这样一个驱动程序的难度超出了一般人的想像。不同的系统对处理缓存一致性上有不同的方法；如果不能正确处理该问题，驱动程序会引起内存冲突。一些系统拥有复杂的总线硬件，使得DMA任务或变得简单，或变得困难。并且不是所有的系统都能对全部的内存执行DMA。幸运的是，内核提供了一个与总线——体系架构无关的DMA层，它会隐藏大多数问题。强烈建议在编写驱动程序时，为DMA操作使用该层。

下面许多函数都需要一个指向`device`结构的指针。该结构是在Linux设备模型中用来表示设备底层的。驱动程序通常不直接使用该结构，但是在使用通用DMA层时，需要使用它。该结构内部隐藏了描述设备的总线细节。比如可以在`pci_device`结构或者`usb_device`结构的`dev`成员中发现它。

使用下列函数的驱动程序都要包含头文件 *<linux/dma-mapping.h>*。

处理复杂的硬件

在执行DMA之前，第一个必须回答的问题是：是否给定的设备在当前主机上具备执行这些操作的能力。出于很多原因，许多设备受限于它们的寻址范围。默认的情况下，内核假设设备都能在32位地址上执行DMA。如果不是这样，应该调用下面的函数通知内核：

```
int dma_set_mask(struct device *dev, u64 mask);
```

该掩码显示与设备能寻址能力对应的位。比如设备受限于24位寻址，则mask应该是0x0FFFFFF。如果使用指定的mask时DMA能正常工作，则返回非零值。如果*dma_set_mask*返回0，则对该设备不能使用DMA。因此，一个受限于24位DMA操作的驱动程序初始化代码有如下的形式：

```
if (dma_set_mask (dev, 0xffffff))
    card->use_dma = 1;
else {
    card->use_dma = 0;
    /* 不得不在没有DMA情况下操作 */
    printk (KERN_WARN, "mydev: DMA not supported\n");
}
```

再强调一遍，如果设备支持常见的32位DMA操作，则没有必要调用*dma_set_mask*。

DMA映射

一个DMA映射是要分配的DMA缓冲区与为该缓冲区生成的、设备可访问地址的组合。我们可以通过对*virt_to_bus*函数的调用获得该地址，但是有许多理由建议不要这么做。第一个理由是具有IOMMU的硬件为总线提供了一套映射寄存器。IOMMU在设备可访问的地址范围内规划了物理内存，使得物理上分散的缓冲区对设备来说变成连续的。对IOMMU的运用需要使用到通用DMA层，而*virt_to_bus*函数不能完成这个任务。

请注意不是所有的体系架构都有IOMMU；特别是常见的x86平台没有对IOMMU的支持。但是，一个正确的驱动程序不需要知道其运行系统上的I/O支持硬件。

在某些情况下，为设备设置可用地址需要建立回弹缓冲区。当驱动程序要试图在外围设备不可访问的地址上执行DMA时（比如高端内存），将创建回弹缓冲区。然后，必要时会将数据写入或者读出回弹缓冲区。对回弹缓冲区的使用势必会降低系统性能，但有的时候却没有其他可替代的方法。

DMA映射必须解决缓存一致性的问题。现代处理器在内部的快速缓存器中保存了最近访问的内存区域；没有该缓存器，将得不到期望的性能。如果设备改变了主内存中的区域，则任何覆盖该区域的处理器缓存都将无效；否则处理器将使用不正确的主内存映射，从而产生不正确的数据。与此类似，当设备使用DMA从主内存中读取数据时，在处理器缓存中的任何改变也必须立刻得到刷新。这些缓存一致性的问题为系统带来诸多不确定因素，如果程序员不细致谨慎的话，这些错误非常难以查找。一些体系架构在硬件中管理缓存的一致性，但是其他一些体系架构则需要软件的支持。通用DMA层竭尽全力来保证在所有体系架构中都能正常运行，但是必须看到，正确的行为需要一套规则来保障。

DMA映射建立了一个新的结构类型——dma_addr_t来表示总线地址。dma_addr_t类型的变量对驱动程序是不透明的；唯一允许的操作是将它们传递给DMA支持例程以及设备本身。作为一个总线地址，如果CPU直接使用了dma_addr_t，将会导致发生不可预期的问题。

根据DMA缓冲区期望保留的时间长短，PCI代码区分两种类型的DMA映射：

一致性DMA映射

这种类型的映射存在于驱动程序生命周期中。一致性映射的缓冲区必须可同时被CPU和外围设备访问（其他类型的映射，如后面将要讨论的类型，在给定时刻只能被一个设备访问）。因此一致性映射必须保存在一致性缓存中。建立和使用一致性映射的开销是很大的。

流式DMA映射

通常为单独的操作建立流式映射。当使用流式映射时，一些体系架构可以最大程度地优化性能，但是这些映射也要服从一组更加严格的访问规则。内核开发者建议尽量使用流式映射，然后再考虑一致性映射。这么做有两个原因。第一个原因是在支持映射寄存器的系统中，每个DMA映射使用总线上的一个或者多个映射寄存器。一致性映射具有很长的生命周期，因此会在相当长的时间内占用这些寄存器，甚至在不使用它们的时候也不释放所有权。第二个原因是在一些硬件中，流式映射可以被优化，但优化的方法对一致性映射无效。

必须用不同的方法操作这两种映射，下面详细描述操作细节。

建立一致性DMA映射

驱动程序可调用*pci_alloc_consistent*函数建立一致性映射：

```
void *dma_alloc_coherent(struct device *dev, size_t size,
                         dma_addr_t *dma_handle, int flag);
```

该函数处理了缓冲区的分配和映射。前两个参数是device结构和所需缓冲区的大小。函数在两处返回DMA映射的结果。函数的返回值是缓冲区的内核虚拟地址，可以被驱动程序使用；而与其相关的总线地址，返回时保存在dma_handle中。该函数对分配的缓冲区做了一些处理，从而缓冲区可用于DMA；通常只是通过*get_free_pages*函数分配内存（请注意size是以字节为单位的，而不是幂次的值）。flag参数通常是描述如何分配内存的GFP_值；通常是GFP_KERNEL或者是GFP_ATOMIC（在原子上下文中运行时）。

当不再需要缓冲区时（通常在模块卸载的时候），调用*dma_free_coherent*向系统返回缓冲区：

```
void dma_free_coherent(struct device *dev, size_t size,
                       void *vaddr, dma_addr_t dma_handle);
```

请注意该函数与其他通用DMA函数一样，需要提供缓冲区大小、CPU地址、总线地址等参数。

DMA池

DMA池是一个生成小型、一致性DMA映射的机制。调用*dma_alloc_coherent*函数获得的映射，可能其最小大小为单个页。如果设备需要的DMA区域比这还小，就要使用DMA池了。在对内嵌于某个大结构中的小型区域执行DMA时，也可以使用DMA池。一些不容易察觉的驱动程序一致性缓存错误，往往存在于结构中与小型DMA区域相邻的成员中。为了避免这一问题的出现，应该总是显式地为DMA操作分配区域，而与其他非DMA数据结构的操作分开。

在*<linux/dmapool.h>*中定义了DMA池的函数。

DMA池必须在使用前，调用下面的函数创建：

```
struct dma_pool *dma_pool_create(const char *name, struct device *dev,
                                 size_t size, size_t align,
                                 size_t allocation);
```

这里，name是DMA池的名字，dev是device结构，size是从该池中分配的缓冲区的大小，align是该池分配操作所必须遵守的硬件对齐原则（用字节表示），如果allocation不为零，表示内存边界不能超越allocation。比如传入的allocation是4096，从该池中分配的缓冲区不能跨越4KB的界限。

当使用完DMA池后，调用下面的函数释放：

```
void dma_pool_destroy(struct dma_pool *pool);
```

在销毁前，必须向DMA池返回所有分配的内存。

使用*dma_pool_alloc*函数处理分配问题：

```
void *dma_pool_alloc(struct dma_pool *pool, int mem_flags,
                     dma_addr_t *handle);
```

在这个函数中，mem_flags通常设置为GFP_ 分配标志。如果一切正常，将分配并返回内存区域（拥有创建DMA池时指定的大小）。像*dma_alloc_coherent*函数一样，返回的DMA缓冲区的地址是内核虚拟地址，并作为总线地址保存在handle中。

使用下面的函数返回不需要的缓冲区：

```
void dma_pool_free(struct dma_pool *pool, void *vaddr, dma_addr_t addr);
```

建立流式DMA映射

由于多种原因，流式映射具有比一致性映射更为复杂的接口。这些映射希望能与已经由驱动程序分配的缓冲区协同工作，因而不得不处理那些不是它们选择的地址。在某些体系架构中，流式映射也能够拥有多个不连续的页和多个“分散/聚集”缓冲区。出于上面这些原因的考虑，流式映射拥有自己的设置函数。

当建立流式映射时，必须告诉内核数据流动的方向。为此定义了一些符号（dma_data_direction枚举类型）：

DMA_TO_DEVICE

DMA_FROM_DEVICE

这两个符号的作用很明显。如果数据被发送到设备（可能使用*write*系统调用作为响应），应使用DMA_TO_DEVICE；而如果数据被发送到CPU，则使用DMA_FROM_DEVICE。

DMA_BIDIRECTIONAL

如果数据可双向移动，则使用DMA_BIDIRECTIONAL。

DMA_NONE

提供该符号只是出于调试目的。如果要使用设置了该符号的缓冲区，会导致内核错误。

可能有的读者认为任何时候都使用DMA_BIDIRECTIONAL就可以了，但是驱动程序作者不能这么做。在一些体系架构中，可能会为这个选择付出很大的性能代价。

当只有一个缓冲区要被传输的时候，使用*dma_map_single*函数映射它：

```
dma_addr_t dma_map_single(struct device *dev, void *buffer, size_t size,
                          enum dma_data_direction direction);
```

返回值是总线地址，可以把它传递给设备；如果执行遇到错误，则返回NULL。

当传输完毕后，使用*dma_unmap_single*函数删除映射：

```
void dma_unmap_single(struct device *dev, dma_addr_t dma_addr, size_t size,
                      enum dma_data_direction direction);
```

在该函数中，size和direction参数必须与映射缓冲区的参数相匹配。

有几条非常重要的原则用于流式DMA映射：

- 缓冲区只能用于这样的传送，即其传送方向匹配于映射时给定的方向值。
- 一旦缓冲区被映射，它将属于设备，而不是处理器。直到缓冲区被撤销映射前，驱动程序不能以任何方式访问其中的内容。只有当*dma_unmap_single*函数被调用后，驱动程序才能安全访问缓冲区中的内容（还存在一个例外，不久就会看到）。尤其要说明的是，这条规则意味着：在包含了所有要写入的数据之前，不能映射要写入设备的缓冲区。
- 在DMA处于活动期间内，不能撤销对缓冲区映射，否则会严重破坏系统的稳定性。

读者可能提出疑问：为什么在缓冲区被映射后，驱动程序不能访问它？有两个原因来解释这条规则。第一，当一个缓冲区建立DMA映射时，内核必须保证在该缓冲区内的全部数据都被写入了内存。当调用*dma_unmap_single*函数时，很可能有一些数据还在处理器的缓存中，因此必须被显式刷新。在刷新动作后，处理器写入缓冲区的数据对设备是不可见的。

第二，如果要映射的缓冲区位于设备不能访问的内存区段时，该怎么办？一些体系架构会只产生一个错误，但是其他一些体系架构将创建一个回弹缓冲区。回弹缓冲区是内存中的独立区域，它可被设备访问。如果使用DMA_TO_DEVICE方向标志映射缓冲区，并且需要使用回弹缓冲区，则在最初缓冲区中的内容作为映射操作的一部分被拷贝。很明显，在拷贝操作后，最初缓冲区内容的改变对设备也是不可见的。同样DMA_FROM_DEVICE回弹缓冲区被*dma_unmap_single*函数拷贝回最初的缓冲区中，也就是说，直到拷贝操作完成，来自设备的数据才可用。

顺便说一下，为什么获得正确的传输方向是一个重要的问题，回弹缓冲区就是一个解释。DMA_BIDIRECTIONAL回弹缓冲区在操作前后都要拷贝数据，这通常会浪费不必要的CPU指令周期。

有时候，驱动程序需要不经过撤销映射就访问流式DMA缓冲区的内容，为此内核提供了如下调用：

```
void dma_sync_single_for_cpu(struct device *dev, dma_handle_t bus_addr,
                             size_t size, enum dma_data_direction direction);
```

应该在处理器访问流式DMA缓冲区前调用该函数。一旦调用了该函数，处理器将“拥有”DMA缓冲区，并可根据需要对它进行访问。然而在设备访问缓冲区前，应该调用下面的函数将所有权交还给设备：

```
void dma_sync_single_for_device(struct device *dev, dma_handle_t bus_addr,
                                size_t size, enum dma_data_direction direction);
```

再次强调，处理器在调用该函数后，不能再访问DMA缓冲区了。

单页流式映射

有时候，要为page结构指针指向的缓冲区建立映射；这种情况是有可能发生的，比如使用*get_user_pages*映射用户空间缓冲区。使用下面的函数，建立和撤销使用page结构指针的流式映射：

```
dma_addr_t dma_map_page(struct device *dev, struct page *page,
                        unsigned long offset, size_t size,
                        enum dma_data_direction direction);

void dma_unmap_page(struct device *dev, dma_addr_t dma_address,
                    size_t size, enum dma_data_direction direction);
```

offset和size参数用于映射一页中的一部分。建议尽量避免映射部分内存页，除非明了其中的原理。如果分配的页是缓存流水线的一部分，则映射部分页会引起一致性问题，比如内存冲突，以及产生非常难以调试的代码缺陷等。

分散/聚集映射

分散/集聚集映射是一种特殊类型的流式DMA映射。假设有几个缓冲区，它们需要与设备双向传输数据。有几种方式能产生这种情形，包括从*readv*或者*writev*系统调用产生，从集群的磁盘I/O请求产生，或者从映射的内核I/O缓冲区中的页面链表产生。可以简单地依次映射每一个缓冲区并且执行请求的操作，但是一次映射整个缓冲区表还是很有利的。

许多设备都能接受一个指针数组的分散表，以及它的长度，然后在一次DMA操作中把它们全部传输走。比如将所有的数据包放在多个数据单元中，“零拷贝”网络非常容易实现。把分散表作为一个整体的另外一个原因是，充分利用那些在总线硬件中含有映射

寄存器系统的优点。在这些系统中，从设备角度上看，物理上不连续的内存页，可以被组装成一个连续数组。这种技术只能用在分散表中的项在长度上等于页面大小的时候（除了第一个和最后一个之外），但是在其工作时，它能够将多个操作转化成单个 DMA 操作，因而能够加速处理工作。

最后，如果必须用到回弹缓冲区，将整个表接合成一个单个缓冲区是很有意义的（因为无论如何它也会被复制）。

所以现在可以确信在某些情况下分散表的映射是值得做的。映射分散表的第一步是建立并填充一个描述被传送缓冲区的scatterlist结构的数组。该结构是与体系架构相关的，并且在头文件 *<linux/scatterlist.h>* 中描述。然而，该结构会始终包含两个成员：

`struct page *page;`

与在 scatter/gather 操作中用到缓冲区相对应的 page 结构指针。

`unsigned int length;`

`unsigned int offset;`

在页内缓冲区的长度和偏移量。

为了映射一个分散/聚集 DMA操作，驱动程序应当为传输的每个缓冲区在scatterlist结构对应入口项上设置 page、offset 和 length 成员。然后调用：

```
int dma_map_sg(struct device *dev, struct scatterlist *sg, int nents,
               enum dma_data_direction direction)
```

这里的nents是传入的分散表入口的数量。返回值是要传送的DMA缓冲区数；它可能会小于 nents。

对在输入分散表中的每一个缓冲区，*dma_map_sg* 函数返回了指定设备的正确的总线地址。作为任务的一部分，它还把内存中相邻的缓冲区接合起来。如果运行驱动程序的系统拥有一个 I/O 内存管理单元，*dma_map_sg* 函数会对该单元的映射寄存器编程，如果没有发生什么错误，则从设备角度上看，其能够传输一块连续的缓冲区。然而在调用之前，是无法知道传输结果的。

驱动程序应该传输由 *dma_map_sg* 函数返回的每个缓冲区。总线地址和每个缓冲区的长度被保存在scatterlist结构中，但是它们在结构中的位置会随体系架构的不同而不同。使用已经定义的两个宏，可用来编写可移植代码：

`dma_addr_t sg_dma_address(struct scatterlist *sg);`

从该分散表的入口项中返回总线（DMA）地址。

```
unsigned int sg_dma_len(struct scatterlist *sg);
```

返回缓冲区的长度。

再次强调，被传输缓冲区的地址和长度与传递给 dma_map_sg 函数的值是不同的。

一旦传输完毕，使用 *dma_unmap_sg* 函数解除分散 / 聚集映射：

```
void dma_unmap_sg(struct device *dev, struct scatterlist *list,
                  int nents, enum dma_data_direction direction);
```

请注意，nents 一定是先前传递给 *dma_map_sg* 函数的入口项的数量，而不是函数返回的 DMA 缓冲区的数量。

分散 / 聚集映射是流式 DMA 映射，因此适用于流式映射的规则也适用于该种映射。如果必须访问映射的分散 / 聚集列表，必须首先对其进行同步：

```
void dma_sync_sg_for_cpu(struct device *dev, struct scatterlist *sg,
                         int nents, enum dma_data_direction direction);
void dma_sync_sg_for_device(struct device *dev, struct scatterlist *sg,
                         int nents, enum dma_data_direction direction);
```

PCI 双重地址周期映射

通常 DMA 支持层使用 32 位总线地址，其为设备的 DMA 掩码所约束。然而 PCI 总线还支持 64 位地址模式，既双重地址周期（DAC）。出于多种原因，通用 DMA 层并不支持该模式，首先这是 PCI 独有的特性。其次，许多 DAC 的实现都是有缺陷的，而且 DAC 也比常用的 32 位 DMA 要慢，会增加性能开销。虽然如此，还是有一些应用程序能正确使用 DAC；如果设备需要使用放在高端内存的大块缓冲区，可以考虑实现 DAC 支持。这种支持只有对 PCI 总线有效，因此必须使用与 PCI 总线相关的例程。

为了使用 DAC，驱动程序必须包含头文件 <*linux/pci.h*>，还必须设置一个单独的 DMA 掩码：

```
int pci_dac_set_dma_mask(struct pci_dev *pdev, u64 mask);
```

只有该函数返回 0 时，才能使用 DAC 地址。

在 DAC 映射中使用了一个特殊类型（dma64_addr_t）。调用 *pci_dac_page_to_dma* 函数建立一个这样的映射：

```
dma64_addr_t pci_dac_page_to_dma(struct pci_dev *pdev, struct page *page,
                                 unsigned long offset, int direction);
```

读者会注意到，可以只使用 page 结构指针（毕竟它们应当保存在高端内存中，否则是

毫无意义的）来建立DAC映射，而且必须以一次一页的方式创建它们。direction参数与在通用DMA层中使用的dma_data_direction枚举类型等价，因此可以取PCI_DMA_TODEVICE、PCI_DMA_FROMDEVICE或者PCI_DMA_BIDIRECTIONAL。

DAC映射不需要其他另外的资源，因此在使用过后，不需要显式释放它。然而像对待其他流式映射一样对待它是必要的，关于缓冲区所有权的规则也适用于它。有一套用于同步DMA缓冲区的函数，其形式如下：

```
void pci_dac_dma_sync_single_for_cpu(struct pci_dev *pdev,
                                     dma64_addr_t dma_addr,
                                     size_t len,
                                     int direction);
void pci_dac_dma_sync_single_for_device(struct pci_dev *pdev,
                                        dma64_addr_t dma_addr,
                                        size_t len,
                                        int direction);
```

一个简单的PCI DMA例子

这里提供了一个PCI设备的DMA例子源代码，以说明如何使用DMA映射。实际PCI总线上的DMA操作形式，与它所驱动的设备密切相关，因此这个例子不能应用于任何真实设备。但它是一个假定的叫*dad*（DMA Acquisition Device，DMA获取设备）驱动程序的一部分。该设备的驱动程序要用类似下面的代码定义传输函数：

```
int dad_transfer(struct dad_dev *dev, int write, void *buffer,
                 size_t count)
{
    dma_addr_t bus_addr;

    /* 映射DMA需要的缓冲区 */
    dev->dma_dir = (write ? DMA_TO_DEVICE : DMA_FROM_DEVICE);
    dev->dma_size = count;
    bus_addr = dma_map_single(&dev->pci_dev->dev, buffer, count,
                              dev->dma_dir);
    dev->dma_addr = bus_addr;

    /* 设置设备 */

    writeb(dev->registers.command, DAD_CMD_DISABLEDMA);
    writeb(dev->registers.command, write ? DAD_CMD_WR : DAD_CMD_RD);
    writel(dev->registers.addr, cpu_to_le32(bus_addr));
    writel(dev->registers.len, cpu_to_le32(count));

    /* 开始操作 */
    writeb(dev->registers.command, DAD_CMD_ENABLEDMA);
    return 0;
}
```

该函数映射了准备进行传输的缓冲区并且启动设备操作。另一半工作必须在中断服务例程中完成，它看起来类似下面这样：

```
void dad_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    struct dad_dev *dev = (struct dad_dev *) dev_id;

    /* 确定该中断确实是从对应的设备发来的 */

    /* 释放对 DMA 缓冲区的映射 */
    dma_unmap_single(dev->pci_dev->dev, dev->dma_addr,
                     dev->dma_size, dev->dma_dir);

    /* 只有到现在这个时候，对缓冲区的访问才是安全的，把它拷贝给用户。 */
    ...
}
```

显而易见，这个例子忽略了大量细节，包括用来阻止同时开始多个 DMA 操作的必要步骤。

ISA 设备的 DMA

ISA 总线允许两种 DMA 传输：本地（native）DMA 和 ISA 总线控制（bus-master）DMA。本地 DMA 使用主板上的标准 DMA 控制器电路来驱动 ISA 总线上的信号线。另一方面，ISA 总线控制 DMA 完全由外围设备控制。后一种 DMA 类型很少被使用，并且也不需要在这里讨论，因为至少从设备角度上看，它与 PCI 设备的 DMA 非常类似。一个 ISA 总线控制 DMA 的例子是 1542 SCSI 控制器，它的驱动程序在内核代码 *drivers/scsi/aha1542.c* 中。

至于这里所关心的本地 DMA，有三种实体涉及到 ISA 总线上的 DMA 数据传输：

8237 DMA 控制器（DMAC）

控制器保存了有关 DMA 传输的信息，比如方向、内存地址、传输数据量大小等。它还包含了一个跟踪传送状态的计数器。当控制器接收到一个 DMA 请求信号时，它将获得总线控制权并驱动信号线，这样设备就能读写数据了。

外围设备

当设备准备传送数据时，必须激活 DMA 请求信号。DMAC 负责管理实际的传输工作；当控制器选通设备后，硬件设备就可以顺序地读写总线上的数据。当传输结束时，设备通常会产生一个中断。

设备驱动程序

需要驱动程序完成的工作很少，它只是负责提供 DMA 控制器的方向、总线地址、

传输量的大小等等。它还与外围设备通信，做好传输数据的准备，当DMA传输完毕后，响应中断。

在PC中使用的早期DMA控制器能够管理四个“通道”，每个通道都与一套DMA寄存器相关联。四个设备可以同时在控制器中保存它们的DMA信息。现在的PC包含了两个与DMAC等价的设备（注6）：第二控制器（主控制器）连接系统的处理器，第一控制器（从控制器）与第二控制器的0通道相连（注7）。

通道编号从0到7：通道4在内部用来将从属控制器级联到主控制器上，因此对ISA外围设备来说，通道4不可用。所以可用的通道是从属控制器（8位通道）上的0～3和主控制器（16位通道）上的5～7。每次DMA传输的大小保存在控制器中，它是一个16位的数，表示传递所需要的总线周期数。因此最大传输大小对从属控制器来说是64KB（因为它在一个周期内传输8位），对主控制器来说是128KB（它使用16位传输）。

因为DMA控制器是系统资源，因此内核协助处理这一资源。内核使用DMA注册表为DMA通道提供了请求/释放机制，并且提供了一组函数在DMA控制器中配置通道信息。

注册 DMA

读者应该对内核注册表很熟悉了，在I/O端口和中断号部分就接触过它。DMA通道注册表与此非常类似。在包含了头文件*<asm/dma.h>*之后，用下面的函数可以获得和释放对DMA通道的所有权：

```
int request_dma(unsigned int channel, const char *name);
void free_dma(unsigned int channel);
```

`channel`参数是0到7的整数，或者更精确地说，是一个小于`MAX_DMA_CHANNELS`的正整数。在PC中，`MAX_DMA_CHANNELS`定义为与硬件匹配的8。`name`参数是标识设备的字符串，指定的名字出现在文件*/proc/dma*中，用户程序可以读取该文件。

*request_dma*函数返回0表示执行成功，返回`-EINVAL`或者`-EBUSY`表示失败。前一个错误码表示所请求的通道超出了范围，后面一个错误码表示通道正为另外一个设备所占用。

注6：这种电路现在是主板芯片组的一部分，但在几年前，它们是两个独立的8237芯片。

注7：最初的PC只有两个控制器，第二个控制器出现在286平台中。但是，第二个控制器被连接为主控制器，这是因为它可以处理16位传输，而第一个控制器每次只能传输八位，因此仅用于向后兼容。

建议读者像对待I/O端口和中断信号线一样地小心处理DMA通道；在*open*操作时请求通道，比在模块初始化函数中请求通道更好一些。延迟请求允许在驱动程序间共享信息；比如声卡和类似的I/O接口如果不在同一时间内使用DMA通道时，就可以共享同一个DMA通道。

这里还建议读者在请求中断信号线之后请求DMA通道，并且在中断之前释放通道。这是在请求两种资源时常用的请求顺序；按照这个顺序可以避免可能的死锁。请注意使用DMA的每个设备还需要一根IRQ线，否则它将无法通知数据已经传输完毕。

在典型应用中，*open*函数的代码有类似以下的形式，这段代码使用了假想的dad模块。*dad*模块不支持共享IRQ信号线，它使用了一个快速中断处理例程。

```
int dad_open (struct inode *inode, struct file *filp)
{
    struct dad_device *my_device;
    /* ... */
    if ( (error = request_irq(my_device.irq, dad_interrupt,
                              SA_INTERRUPT, "dad", NULL)) )
        return error;  /* 或者实现阻塞的open操作 */

    if ( (error = request_dma(my_device.dma, "dad")) ) {
        free_irq(my_device.irq, NULL);
        return error; /* 或者实现阻塞的open操作 */
    }
    /* ... */
    return 0;
}
```

与*open*函数相匹配的*close*函数的实现如下：

```
void dad_close (struct inode *inode, struct file *filp)
{
    struct dad_device *my_device;

    /* ... */
    free_dma(my_device.dma);
    free_irq(my_device.irq, NULL);
    /* ... */
}
```

下面是一个安装了声卡系统的*/proc/dma*文件：

```
merlino% cat /proc/dma
 1: Sound Blaster8
 4: cascade
```

请注意默认的声卡驱动程序在系统启动时获得了DMA通道，并且不会释放它。cascade入口是一个占位符，表示通道4对驱动程序不可用，其原因在前面讲述过了。

与DMA控制器通信

注册之后，驱动程序的主要任务包括为适当的操作配置DMA控制器。该任务不是可有可无的，但幸运的是，内核导出了驱动程序所需要的所有函数。

当*read*或者*write*函数被调用时，或者准备异步传输时，驱动程序都要对DMA控制器进行配置。根据驱动程序和其实现配置的策略，这一工作可以在调用*open*函数时，或者响应*ioctl*命令时被执行。这里的代码是被*read*和*write*设备方法调用的典型代码。

这一小节提供了DMA控制器内部的概貌，这样，读者就能理解这里所给出的代码。如果想要了解更多信息，可阅读*<asm/dma.h>*文件以及描述PC体系架构的硬件手册。特别是在这里不处理与16位数据传输相对的8位传输问题。如果要为ISA设备板卡编写驱动程序，应该仔细阅读该设备的硬件手册，以查找相关信息。

DMA通道是一个可共享的资源，如果多于一个的处理器要同时对其进行编程，则会产生冲突。因此有一个叫作dma_spin_lock的自旋锁保护控制器。驱动程序不能直接处理该自旋锁；但是有两个函数能够对它进行操作：

`unsigned long claim_dma_lock();`

获得DMA自旋锁。该函数也阻塞本地处理器上的中断，因此返回值是描述先前中断状态的一系列标志。在重新开中断时，返回值必须传递给下面的函数以恢复中断状态。

`void release_dma_lock(unsigned long flags);`

返回DMA自旋锁并恢复先前的中断状态。

当使用下面描述的函数时，应该拥有自旋锁。但是在实际的I/O中却不应该拥有自旋锁。当拥有自旋锁时，驱动程序不能处于休眠状态。

必须被装入控制器的信息包含三个部分：RAM的地址、必须被传输的原子项个数（以字节或字为单位）以及传输的方向。为达到这个目的，*<asm/dma.h>*定义了下面的函数：

`void set_dma_mode(unsigned int channel, char mode);`

表明是从设备读入通道（DMA_MODE_READ）还是向设备写入数据（DMA_MODE_WRITE）。还有第三个模式——DMA_MODE_CASCADE，它用来释放对总线的控制。级联是将第一控制器连接到第二控制器顶端的方法，但是也可以用于真正的ISA总线控制设备。在这里不讨论总线控制。

`void set_dma_addr(unsigned int channel, unsigned int addr);`

为DMA缓冲区分配地址。该函数将addr的最低24位存储到控制器中。addr参数必须是总线地址（请参看本章前面的“总线地址”部分）。

`void set_dma_count(unsigned int channel, unsigned int count);`

为传输的字节量赋值。count 参数也可以表示16位通道的字节数，此时字节数必须是偶数。

除了这些函数外，当处理 DMA 设备时，还有许多用于管理设备的函数：

`void disable_dma(unsigned int channel);`

控制器内的DMA通道可以被禁用。应该在配置控制器前禁用通道，以防止不正确的操作（否则由于控制器是针对8位数据传输编程的，可能会引起冲突，因此前面所介绍的函数都不能原子地执行）。

`void enable_dma(unsigned int channel);`

该函数告诉控制器，DMA 通道中包含了合法的数据。

`int get_dma_residue(unsigned int channel);`

驱动程序有时需要知道 DMA 传输是否已经结束。该函数返回还未传输的字节数。如果传输成功，返回值是0，当控制器正在工作时，返回值并不确定（但不会是0）。如果使用两个 8 位输入操作来获得一个 16 位的余量，则返回值是不可预测的。

`void clear_dma_ff(unsigned int channel)`

该函数清除了DMA的触发器（flip-flop）。触发器用于控制对16位寄存器的访问。我们可以通过两个连续的8位操作访问该寄存器，而触发器被用来选择低字节（当其清零时）还是高字节（当其被置位）。当传输完 8 位后，触发器自动反转；程序员必须在访问 DMA 寄存器前清除触发器（将其设置为可知状态）。

使用这些函数，驱动程序可以实现如下函数，为 DMA 传输做准备：

```
int dad_dma_prepare(int channel, int mode, unsigned int buf,
                    unsigned int count)
{
    unsigned long flags;

    flags = claim_dma_lock();
    disable_dma(channel);
    clear_dma_ff(channel);
    set_dma_mode(channel, mode);
    set_dma_addr(channel, virt_to_bus(buf));
    set_dma_count(channel, count);
    enable_dma(channel);
    release_dma_lock(flags);

    return 0;
}
```

接着，下面的代码用来检查是否成功完成 DMA：

```
int dad_dma_isdone(int channel)
{
    int residue;
    unsigned long flags = claim_dma_lock ();
    residue = get_dma_residue(channel);
    release_dma_lock(flags);
    return (residue =  = 0);
}
```

剩下需要做的唯一事情是配置设备板卡。这个与设备相关的任务通常包括读写一些I/O端口。不同设备实现的方法差异很大。比如一些设备希望程序员告诉硬件DMA缓冲区的大小，而有些时候驱动程序不得不读出固化在设备中的数据。为了完成对板卡的配置，硬件手册是程序员唯一的朋友。

快速参考

本章介绍了下面这些与内存操作相关的符号。

介绍材料

```
#include <linux/mm.h>
#include <asm/page.h>
```

在这些头文件中定义了大部分与内存管理相关的函数和结构，并给出了原型。

```
void *_ _va(unsigned long physaddr);
unsigned long _ _pa(void *kaddr);
```

在内核逻辑地址和物理地址之间进行转换的宏。

```
PAGE_SIZE
PAGE_SHIFT
```

前者以字节为单位给出在特定硬件中每页的大小，后者表示为获得物理地址，页帧号需要移动的位数。

```
struct page
```

在系统内存映射中，表示硬件页的结构。

```
struct page *virt_to_page(void *kaddr);
void *page_address(struct page *page);
struct page *pfn_to_page(int pfn);
```

负责在内核逻辑地址和与其相关的内存映射入口之间进行转换的宏。*page_address*只能对已经显式映射的低端内存或者高端内存进行操作。*pfn_to_page*将页帧号转换为与其相关的page结构指针。

```
unsigned long kmap(struct page *page);
void kunmap(struct page *page);
```

*kmap*返回映射到指定页的内核虚拟地址，如果需要的话，还创建映射。*kunmap*为指定页删除映射。

```
#include <linux/highmem.h>
#include <asm/kmap_types.h>
void *kmap_atomic(struct page *page, enum km_type type);
void kunmap_atomic(void *addr, enum km_type type);
```

*kmap*的高性能版本；只有原子代码才能拥有映射结果。对驱动程序来说，type可以是KM_USER0、KM_USER1、KM_IRQ0或者是KM_IRQ1。

```
struct vm_area_struct;
```

描述VMA的结构。

mmap的实现

```
int remap_pfn_range(struct vm_area_struct *vma, unsigned long virt_add,
  unsigned long pfn, unsigned long size, pgprot_t prot);
int io_remap_page_range(struct vm_area_struct *vma, unsigned long virt_add,
  unsigned long phys_add, unsigned long size, pgprot_t prot);
```

*mmap*的核心函数。它们映射了物理地址中从pfn表示的页号开始的size个字节，到虚拟地址virt_add上。相关虚拟地址的保护位在prot中指定。如果目标地址是在I/O地址空间的话，使用*io_remap_page_range*函数。

```
struct page *vmalloc_to_page(void *vmaddr);
```

将从*vmalloc*函数返回的内核虚拟地址转化为相对应的page结构指针。

直接I/O的实现

```
int get_user_pages(struct task_struct *tsk, struct mm_struct *mm, unsigned
  long start, int len, int write, int force, struct page **pages, struct
  vm_area_struct **vmas);
```

该函数将用户空间缓冲区锁进内存，并返回相应的page结构指针。调用者必须拥有mm->mmap_sem。

```
SetPageDirty(struct page *page);
```

将指定内存页标记为“已经改动过”，并在释放该页前将其写入后备存储器的宏。

```
void page_cache_release(struct page *page);
```

从页缓存中释放指定的页。

```
int is_sync_kiocb(struct kiocb *iocb);
```

如果指定的IOCB需要同步执行，该宏返回非零值。

```
int aio_complete(struct kiocb *iocb, long res, long res2);
```

表明异步I/O操作完成的函数。

直接内存访问

```
#include <asm/io.h>
unsigned long virt_to_bus(volatile void * address);
void * bus_to_virt(unsigned long address);
```

用来在内核、虚拟地址、总线地址之间进行转换的老版本函数。与外围设备通信时，必须使用总线地址。

```
#include <linux/dma-mapping.h>
```

用来定义通用DMA函数的头文件。

```
int dma_set_mask(struct device *dev, u64 mask);
```

对于那些不能在全部32位寻址的外围设备来说，该函数通知内核可寻址的范围，如果DMA可行则返回非零值。

```
void *dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t
  *bus_addr, int flag)
void dma_free_coherent(struct device *dev, size_t size, void *cpuaddr,
  dma_handle_t bus_addr);
```

为缓冲区分配和释放一致性DMA映射，该缓冲区的生命周期与驱动程序相同。

```
#include <linux/dmapool.h>
struct dma_pool *dma_pool_create(const char *name, struct device *dev,
  size_t size, size_t align, size_t allocation);
void dma_pool_destroy(struct dma_pool *pool);
void *dma_pool_alloc(struct dma_pool *pool, int mem_flags, dma_addr_t
   *handle);
void dma_pool_free(struct dma_pool *pool, void *vaddr, dma_addr_t handle);
```

用来创建、销毁和使用DMA池的函数，用来管理小型的DMA区域。

```
enum dma_data_direction;
DMA_TO_DEVICE
DMA_FROM_DEVICE
DMA_BIDIRECTIONAL
DMA_NONE
```

用来告诉流式映射函数的符号，表明数据传输的方向。

```
dma_addr_t dma_map_single(struct device *dev, void *buffer, size_t size, enum
  dma_data_direction direction);
void dma_unmap_single(struct device *dev, dma_addr_t bus_addr, size_t size,
  enum dma_data_direction direction);
```

创建和销毁单个流式 DMA 映射。

```
void dma_sync_single_for_cpu(struct device *dev, dma_handle_t bus_addr, size_t
  size, enum dma_data_direction direction);
void dma_sync_single_for_device(struct device *dev, dma_handle_t bus_addr,
  size_t size, enum dma_data_direction direction);
```

同步拥有流式映射的缓冲区。在使用流式映射时，如果处理器要访问缓冲区，则必须使用这些函数。

```
#include <asm/scatterlist.h>
struct scatterlist { /* ... */ };
dma_addr_t sg_dma_address(struct scatterlist *sg);
unsigned int sg_dma_len(struct scatterlist *sg);
```

scatterlist结构描述了包含多个缓冲区的I/O操作。当实现了分散/聚集操作时，宏*sg_dma_address*和*sg_dma_len*用来获得总线地址和缓冲区长度，并将它们传递给设备。

```
dma_map_sg(struct device *dev, struct scatterlist *list, int nents,
  enum dma_data_direction direction);
dma_unmap_sg(struct device *dev, struct scatterlist *list, int nents, enum
  dma_data_direction direction);
void dma_sync_sg_for_cpu(struct device *dev, struct scatterlist *sg, int
  nents, enum dma_data_direction direction);
void dma_sync_sg_for_device(struct device *dev, struct scatterlist *sg, int
  nents, enum dma_data_direction direction);
```

dma_map_sg 函数映射了分散 / 聚集操作，*dma_unmap_sg* 函数负责解除映射。当映射处于活动状态而又必须访问缓冲区时，需要使用 *dma_sync_sg_** 函数进行同步。

/proc/dma

包含DMA控制器中被分配通道信息快照的文本文件。其中不会显示基于PCI的DMA，因为它们独立工作，不需要在DMA控制器中分配一个通道。

#include <asm/dma.h>

定义与DMA相关的函数和宏的头文件，其中给出了函数原型。如果要使用下面的符号，必须包含该文件。

```
int request_dma(unsigned int channel, const char *name);
void free_dma(unsigned int channel);
```

访问DMA注册表。在使用ISA DMA通道前必须执行注册。

```
unsigned long claim_dma_lock();
void release_dma_lock(unsigned long flags);
```

获得和释放DMA自旋锁，在调用ISA DMA函数前必须拥有自旋锁。它们也能禁止和重新打开本地处理器的中断。

```
void set_dma_mode(unsigned int channel, char mode);
void set_dma_addr(unsigned int channel, unsigned int addr);
void set_dma_count(unsigned int channel, unsigned int count);
```

在DMA控制器内对DMA信息编程。addr是总线地址。

```
void disable_dma(unsigned int channel);
void enable_dma(unsigned int channel);
```

在配置时，DMA通道必须被禁止。这些函数用来改变DMA通道状态。

```
int get_dma_residue(unsigned int channel);
```

如果驱动程序需要知道DMA传输的进展情况，可以调用这个函数，它将返回未被传输数据的数量。在成功完成DMA后，该函数返回0；当数据正在被传输时，返回值是不可预测的。

```
void clear_dma_ff(unsigned int channel)
```

DMA控制器使用触发器并通过两个8位操作传输一个16位的值。在把数据传递给控制器前，必须清除触发器。

第十六章

块设备驱动程序



迄今为止，我们的论述只局限于字符设备驱动程序，然而在Linux系统中还有另外一类驱动程序，现在是该把目光转向这类设备的时候了。本章将讨论块设备驱动程序。

一个块设备驱动程序主要通过传输固定大小的随机数据来访问设备。Linux内核视块设备为与字符设备相异的基本设备类型，因此块驱动程序有自己完成特定任务的接口。

高效的块设备驱动程序在性能上是严格要求的，并不仅仅体现在用户应用程序的读、写操作中。现代操作系统使用虚拟内存工作，把不需要的数据转移到诸如磁盘等其他存储介质上。块驱动程序是在核心内存与其他存储介质之间的管道，因此它们可以认为是虚拟内存子系统的组成部分。虽然不需要知道页结构和其他重要的内存概念就能编写出块设备驱动程序，但是为了编写出高性能的驱动程序，编码人员还是需要了解在第十五章讲述的内容。

对块设备分层的设计，其着眼点是性能。许多字符设备可以在远低于其最快速率下工作，因此，对系统综合性能的影响并不显著。但如果这种问题出现在块I/O子系统中，系统就不能正常工作了。Linux块设备驱动程序接口使得块设备可以发挥其最大的功效，但是其复杂程度又是编程者必须面对的一个问题。所幸的是2.6版本的块设备接口对过去版本的内核做了很大修改和提高。

正如读者希望的一样，本章提供了一个例子驱动程序，其实现了基于内存的块设备驱动程序。本质上讲，这是一个ramdisk（内存盘）驱动程序。内核已经包含了更高级的ramdisk的实现，但是例子驱动程序（名为*sbull*）在尽量减少相关复杂度的前提下，告诉读者如何创建一个块设备驱动程序。

在深入细节以前，先精确定义一些术语。一个数据块指的是固定大小的数据，而大小的值由内核确定。数据块的大小通常是4096个字节，但是可以根据体系结构和所使用的文件系统进行改变。与数据块对应的是扇区，它是由底层硬件决定大小的一个块。内核所

处理的设备扇区大小是512字节。如果用户的设备使用了不同的大小，需要对内核进行修改，以避免产生硬件所不能处理的I/O请求。要记住这一点：无论何时内核为用户提供了一个扇区编号，该扇区的大小就是512字节。如果要使用不同的硬件扇区大小，用户必须对内核的扇区数做相应的修改。在*sbull*驱动程序中将演示这一过程。

注册

与字符设备一样，块设备必须使用一系列的注册函数，使内核知道设备的存在。概念虽然十分相似，但是块设备的注册细节是截然不同的。读者需要学习一套新的数据结构和设备操作。

注册块设备驱动程序

对大多数块设备驱动程序来说，第一步是向内核注册自己。执行该任务的函数是*register_blkdev*（在*<linux/fs.h>*中声明）：

```
int register_blkdev(unsigned int major, const char *name);
```

参数是该设备使用的主设备号及其名字（内核在*/proc/devices*中显示的名字）。如果传递的主设备号是0，内核将分派一个新的主设备号给设备，并将该设备号返回给调用者。如果调用*register_blkdev*返回负值，则表示出现了一个错误。

与其对应的注销块设备驱动程序的函数是：

```
int unregister_blkdev(unsigned int major, const char *name);
```

这里，参数必须与传递给*register_blkdev*的参数相匹配，否则函数将返回-EINVAL，并且不做任何注销工作。

在内核2.6中，对*register_blkdev*的调用是完全可选的。函数*register_blkdev*所执行的功能随时间的推移而越来越少；在这里，该接口所做的事情是：其一，如果需要的话分配一个动态的主设备号；其二，在*/proc/devices*中创建一个入口项。在未来的内核中，可能会取消*register_blkdev*函数。但是在这段时间内，大多数驱动程序依然会调用该函数，因为这是个传统。

注册磁盘

虽然*register_blkdev*能获得主设备号，但是它并不能让系统使用任何磁盘。因此为了管理独立的磁盘，必须使用另外一个单独的注册接口。使用该接口需要熟悉一系列的新的数据结构，而这是学习的起点。

块设备操作

字符设备使用file_operations结构告诉系统对它们的操作接口。块设备使用类似的数据结构；在<*linux/fs.h*>中声明了结构block_device_operations。下面的内容就是对该结构中的各个成员的简要说明，当以后讲到*sbull*驱动程序细节的时候，还要对它们做更详细的介绍：

```
int (*open)(struct inode *inode, struct file *filp);
int (*release)(struct inode *inode, struct file *filp);
```

与字符设备中对应函数功能作用相同，当设备被打开或者关闭时调用它们。一个块设备驱动程序可能用旋转盘片、锁住仓门（对可移动介质）等来响应open调用。如果用户将介质放入设备中锁住，那么在*release*函数中当然要进行解锁。

```
int (*ioctl)(struct inode *inode, struct file *filp, unsigned int cmd,
  unsigned long arg);
```

实现*ioctl*系统调用的函数。块设备层会首先截取大量的标准请求，因此大多数块设备的*ioctl*函数都十分短小。

```
int (*media_changed) (struct gendisk *gd);
```

内核调用该函数以检查用户是否更换了驱动器内的介质，如果用户更换了，那么返回一非零值。很明显，该函数只适用于那些支持可移动介质（并且为驱动程序设置了“介质更换”标志位）的驱动器；在其他情况下，该函数被忽略。

内核使用gendisk结构表示一个独立的磁盘，在下一节中将要对该结构进行讲解。

```
int (*revalidate_disk) (struct gendisk *gd);
```

当介质被更换时，调用*revalidate_disk*函数做出响应；它告诉驱动程序完成必要的工作，以便使用新的介质。该函数返回一个int型值，但是该值为内核所忽略。

```
struct module *owner;
```

一个指向拥有该结构的模块指针，通常它都被初始化为THIS_MODULE;

在这个列表中，细心的读者可能会发现一个有趣的现象：没有函数负责读和写数据。在块设备的I/O子系统中，这些操作是由*request*函数处理的。*request*函数能完成众多操作，在本章后面的部分将对该函数进行讨论。在讨论请求服务前，必须完成对磁盘注册的讨论。

gendisk结构

内核使用gendisk结构（在<*linux/genhd.h*>中声明）来表示一个独立的磁盘设备。实际上，内核还使用gendisk结构表示分区，但是驱动程序的作者并不需要了解这些。在gendisk结构中的许多成员必须由驱动程序进行初始化：

```
int major;
int first_minor;
int minors;
```

磁盘使用这些成员描述设备号。一个驱动器至少使用一个次设备号。如果驱动器是可被分区的（大多数情况下），用户将要为每个可能的分区都分配一个次设备号。minors成员常取16，它使得一个“完整的磁盘”可以包含15个分区。但是，某些磁盘驱动程序设置每个设备可使用64个次设备号。

```
char disk_name[32];
```

设置磁盘设备名字的成员。该名字将显示在*/proc/partitions*和sysfs中。

```
struct block_device_operations *fops;
```

设置前面表述的各种设备操作；

```
struct request_queue *queue;
```

内核使用该结构为设备管理I/O请求；在“请求过程”一节中将对其进行论述。

```
int flags;
```

用来描述驱动器状态的标志（很少使用）。如果用户设备包含了可移动介质，其将被设置为GENHD_FL_REMOVABLE。CD-ROM设备被设置为GENHD_FL_CD。如果出于某些原因，不希望在*/proc/partitions*中显示分区信息，则可将该标志设为GENHD_FL_SUPPRESS_PARTITION_INFO。

```
sector_t capacity;
```

以512字节为一个扇区时，该驱动器可包含的扇区数。sector_t可以是64位宽。驱动程序不能直接设置该成员，而要将扇区数传递给*set_capacity*。

```
void *private_data;
```

块设备驱动程序可能使用该成员保存指向其内部数据的指针。

内核为使用gendisk结构提供了一些函数，下面介绍这些函数，然后将看到*sbull*如何使用它们，从而让它的磁盘设备为系统服务。

gendisk结构是一个动态分配的结构，它需要一些内核的特殊处理来进行初始化；驱动程序不能自己动态分配该结构，而是必须调用：

```
struct gendisk *alloc_disk(int minors);
```

参数minors是该磁盘使用的次设备号的数目。请注意，为了能正常工作，此后用户就不能改变 minors 成员。

当不再需要一个磁盘时，调用下面的函数卸载磁盘：

```
void del_gendisk(struct gendisk *gd);
```

gendisk是一个引用计数结构（它包含一个kobject对象）。*get_disk*和*put_disk*函数负责处理引用计数，但是驱动程序不能直接使用这两个函数。通常对*del_gendisk*的调用会删除gendisk中的最终计数，但是没有机制能保证其肯定发生。因此当调用*del_gendisk*后，该结构可能继续存在（而且内核可能会调用我们提供的各种方法）。当没有用户继续使用时（当最终被释放、或者在模块的*cleanup*函数中），将真正删除该结构，此后，我们可以肯定不会再得到任何该设备的信息了。

分配一个gendisk结构并不能使磁盘对系统可用。为达到这个目的，必须初始化结构并调用*add_disk*:

```
void add_disk(struct gendisk *gd);
```

请记住这个关键点：一旦调用了*add_disk*，磁盘设备将被“激活”，并随时会调用它提供的方法。实际上第一次对这些方法的调用可能在*add_disk*返回前就发生了，这是因为，内核可能会读取前面几个块的数据以获得分区表。因此在驱动程序完全被初始化并且能够响应对磁盘的请求前，请不要调用*add_disk*。

sbull中的初始化

现在是仔细研究例子程序的时候了。*sbull*驱动程序（该示例与其他例子的源代码可以在O'Reilly的FTP站点上获得）实现了一个内存虚拟磁盘驱动器。对于每个驱动器来说，*sbull*分配（为简化起见，使用了*vmalloc*）了一个内存数组，然后通过块设备操作使该数组可用。我们可以通过对这个虚拟设备进行分区，在其上建立文件系统，并把它挂装到文件系统树上来测试*sbull*驱动程序。

与其他例子驱动程序相同，*sbull*允许在编译或者加载的时候获得主设备号。如果没有指定主设备号，则将动态分配一个。由于动态分配需要调用*register_blkdev*函数，所以*sbull*的代码如下:

```
sbull_major = register_blkdev(sbull_major, "sbull");
if (sbull_major <= 0) {
    printk(KERN_WARNING "sbull: unable to get major number\n");
    return -EBUSY;
 }
```

与本书中的其他虚拟设备相同，*sbull*设备也用内部的一个数据结构来描述:

```
struct sbull_dev {
        int size;                       /* 以扇区为单位，设备的大小 */
        u8 *data;                       /* 数据数组 */
        short users;                    /* 用户数目 */
```

```
        short media_change;           /* 介质改变标志 */
        spinlock_t lock;              /* 用于互斥 */
        struct request_queue *queue;  /* 设备请求队列 */
        struct gendisk *gd;           /* gendisk结构 */
        struct timer_list timer;      /* 用来模拟介质改变 */
};
```

需要通过几个步骤来初始化该结构，使系统能操控相应的设备。现在从基本的初始化和分配底层内存开始讲解：

```
memset (dev, 0, sizeof (struct sbull_dev));
dev->size = nsectors*hardsect_size;
dev->data = vmalloc(dev->size);
if (dev->data =  = NULL) {
    printk (KERN_NOTICE "vmalloc failure.\n");
    return;
}
spin_lock_init(&dev->lock);
```

在进行下一步分配请求队列前，对自旋锁进行分配和初始化非常重要。在进行请求处理时，再仔细分析这个过程。现在为了满足这个要求所必须的步骤是：

```
dev->queue = blk_init_queue(sbull_request, &dev->lock);
```

这里的*sbull_request*是请求函数，它负责执行块设备的读、写请求。当分配了一个请求队列时，必须提供一个自旋锁用以控制对队列的访问。该锁由驱动程序而不是内核提供的原因是：请求队列和其他驱动程序的数据结构常常会落到同一个临界区内，它们将要被同时访问。和其他任意一个分配内存的函数一样，*blk_init_queue*可能会失败，因此在进行下一步之前，必须检查其返回值。

一旦在适当的位置拥有了设备内存和请求队列，就可以分配、初始化及安装相应的gendisk结构了。用下面的代码执行该任务：

```
dev->gd = alloc_disk(SBULL_MINORS);
if (! dev->gd) {
    printk (KERN_NOTICE "alloc_disk failure\n");
    goto out_vfree;
}
dev->gd->major = sbull_major;
dev->gd->first_minor = which*SBULL_MINORS;
dev->gd->fops = &sbull_ops;
dev->gd->queue = dev->queue;
dev->gd->private_data = dev;
snprintf (dev->gd->disk_name, 32, "sbull%c", which + 'a');
set_capacity(dev->gd, nsectors*(hardsect_size/KERNEL_SECTOR_SIZE));
add_disk(dev->gd);
```

这里，SBULL_MINORS是每个*sbull*设备所支持的次设备号的数量。当为每一个设备设

置它的次设备号时，都必须考虑到那些已经分配给设备的其他次设备号。被设置的设备名第一个是*sbulla*，第二个是*sbullb*。用户空间可以添加分区数，这样在第二个设备上的第三个分区可能是*/dev/sbullb3*。

最后调用*add_disk*结束设置过程。由于在*add_disk*返回前，有可能会调用其他的一些磁盘操作函数，因此对*add_disk*的调用一定要放在初始化设备的最后一步。

对扇区大小的说明

正如前面所提到的，内核认为每个磁盘都是由512字节大小的扇区所组成的线形数组。但并不是所有硬件都使用这个扇区大小的。让一个具有不同扇区大小的设备运行起来并不是特别困难，只是仔细关心许多细节。*sbull*设备输出了一个`hardsect_size`参数，使用该参数可以改变“硬件”设备扇区的大小；参看代码的实现，读者就能知道如何在自己的设备中加入该种支持。

所有操作的第一步就是通知内核设备所支持的扇区大小。硬件扇区大小作为一个参数放在请求队列中，而不是放在`gendisk`结构中。当分配队列之后，立刻调用*blk_queue_hardsect_size*设置扇区大小：

```
blk_queue_hardsect_size(dev->queue, hardsect_size);
```

当进行完上述调用后，内核就对我们的设备使用设定的硬件扇区大小。所有的I/O请求都将定位在硬件扇区的开始位置，并且每个请求的大小都将是扇区大小的整数倍。必须记住，内核总是认为扇区大小是512字节，因此必须将所有的扇区数进行转换。比如当*sbull*在自己的`gendisk`结构中设置硬件容量时，调用如下代码：

```
set_capacity(dev->gd, nsectors*(hardsect_size/KERNEL_SECTOR_SIZE));
```

`KERNEL_SECTOR_SIZE`是本地定义的一个常量，使用该常量进行内核512字节扇区到实际使用扇区大小的转换。我们会在*sbull*的请求处理过程看到很多这种类型的计算。

块设备操作

在前面的部分已经对`block_device_operations`结构进行了简要的介绍。在进入请求处理部分之前，我们要花一点时间仔细研究这些操作。在本节的最后，再来讨论*sbull*驱动程序的一个新功能：它把自己伪装成一个可移动的设备。无论何时当最后一个用户关闭了设备，都要设置一个30秒的定时器；如果在这个时段内设备没有被打开，设备中的所有内容将被清除，内核将被告知设备介质已经被改变了。30秒的延迟给用户提供了一个机会，比如，在设备上创建文件系统之后再挂装*sbull*设备。

open 和 release 函数

为了模拟移动介质，*sbull*必须知道最后一个用户何时关闭了设备。驱动程序维护了一个用户计数。*open* 和 *close* 函数的一个任务就是更新用户计数。

*open*函数与字符设备中的*open*函数很类似；它把相关的inode和file结构作为参数。当一个inode指向一个块设备时，i_bdev->bd_disk成员包含了指向相应gendisk结构的指针；该指针可用于获得驱动程序内部的数据结构。实际上，*sbull*的*open*函数是这么做的：

```
static int sbull_open(struct inode *inode, struct file *filp)
{
    struct sbull_dev *dev = inode->i_bdev->bd_disk->private_data;

    del_timer_sync(&dev->timer);
    filp->private_data = dev;
    spin_lock(&dev->lock);
    if (! dev->users)
        check_disk_change(inode->i_bdev);
    dev->users++;
    spin_unlock(&dev->lock);
    return 0;
}
```

当*sbull_open*拥有自己的设备结构指针后，此时如果有任何处于活动状态的“介质移除”定时器，它将调用*del_timer_sync*删除“介质移除”定时器。请注意在定时器被删除前，没有锁定设备的自旋锁；如果锁定的话，一旦在删除定时器前执行了定时器函数，则会造成死锁。锁定设备后，调用内核函数*check_disk_change*检查介质是否改变。可能有的读者会认为内核应该自己调用该函数，但是标准的模式是由驱动程序在*open*的时候处理它。

最后一步是增加用户计数并返回。

与此相反，*release*函数的功能是：减少用户计数并启动“介质移除”定时器：

```
static int sbull_release(struct inode *inode, struct file *filp)
{
    struct sbull_dev *dev = inode->i_bdev->bd_disk->private_data;

    spin_lock(&dev->lock);
    dev->users--;

    if (!dev->users) {
        dev->timer.expires = jiffies + INVALIDATE_DELAY;
        add_timer(&dev->timer);
    }
    spin_unlock(&dev->lock);
```

```
    return 0;
}
```

对于那些操作实际硬件设备的驱动程序，*open*和*release*函数可以设置驱动程序和硬件的状态。这些操作包括使磁盘开始或者停止旋转，锁住可移动介质的仓门，以及分配DMA缓存等等。

读者可能会问：到底是谁真正打开了块设备？有一些操作能够让块设备在用户空间内被直接打开，这些操作包括给磁盘分区，或者在分区上创建文件系统，或者运行文件系统检查程序。当挂装一个分区时，块设备驱动程序也会调用*open*函数。此时没有用户空间进程拥有设备的文件描述符，而打开的文件为内核自己所保持。一个块设备驱动程序无法区分挂装操作（在内核空间内打开设备）与诸如*mkfs*这样的应用程序（在用户空间中打开设备）调用之间的区别。

对可移动介质的支持

block_device_operations结构包含了两个函数用以支持移动介质。如果读者是为非移动设备编写驱动程序，则可以忽略这两个函数。这两个函数的实现是相当直接的。

（从*check_disk_change*函数中）调用*media_changed*函数以检查介质是否被改变；如果被改变则该函数将返回非零值。*sbull*中的实现很简单：如果介质移除定时器到期，将设置一个标志位，*sbull*仅仅查询该标志位：

```
int sbull_media_changed(struct gendisk *gd)
{
    struct sbull_dev *dev = gd->private_data;

    return dev->media_change;
}
```

在介质改变后将调用*revalideate*函数；为了让驱动程序能操作新的介质，该函数要完成所有必需的工作。调用*revalidate*后，内核将试着重新读取设备的分区表。在*sbull*的实现中只是简单地重新设置了media_change标志位，并清除了设备内存空间以模拟插入一张空白磁盘。

```
int sbull_revalidate(struct gendisk *gd)
{
    struct sbull_dev *dev = gd->private_data;

    if (dev->media_change) {
        dev->media_change = 0;
        memset (dev->data, 0, dev->size);
    }
    return 0;
}
```

ioctl 函数

块设备驱动程序提供了*ioctl*函数执行设备的控制功能。高层的块设备子系统在驱动程序获得*ioctl*命令前，已经截取了大量的命令（请参看内核代码*drivers/block/ioctl.c*对该过程进行全面了解）。实际上在一个现代驱动程序中，许多*ioctl*命令根本就不用实现。

*sbull*的*ioctl*函数只处理一个命令 —— 对设备物理信息的查询请求：

```
int sbull_ioctl (struct inode *inode, struct file *filp,
                 unsigned int cmd, unsigned long arg)
{
    long size;
    struct hd_geometry geo;
    struct sbull_dev *dev = filp->private_data;

    switch(cmd) {
        case HDIO_GETGEO:
        /*
         * 获得物理信息：由于是虚拟设备，因此不得不提供一些虚拟的信息。
         * 因此这里声明有16个扇区、4个磁头，并且计算相应的柱面数。
         * 这里，我们设置数据开始的位置在第四扇区。
         */
        size = dev->size*(hardsect_size/KERNEL_SECTOR_SIZE);
        geo.cylinders = (size & ~0x3f) >> 6;
        geo.heads = 4;
        geo.sectors = 16;
        geo.start = 4;
        if (copy_to_user((void __user *) arg, &geo, sizeof(geo)))
            return -EFAULT;
        return 0;
    }

    return -ENOTTY; /* 未知命令 */
}
```

由于例子程序的设备是一个纯粹的虚拟设备，并且没有磁道和柱面，因此提供设备的物理参数信息显得很奇怪。多年来大多数实际的块设备硬件都是用非常复杂的数据结构来描述的。内核对块设备的物理信息并不感兴趣，它只把设备看成是线性的扇区数组。但是一些用户空间的应用程序依然需要查询磁盘的物理信息。特别是*fdisk*工具，它负责根据柱面信息编辑分区表，如果这些信息获得不了，它将不能正常工作。

我们希望使用那些古老且简单的工具就能对*sbull*设备进行分区，因此，*ioctl*函数提供了可靠的虚拟物理参数与设备相匹配。大多数磁盘驱动程序所做的工作与此类似。请注意，如果需要的话，扇区数目要与内核使用的512字节约定相匹配。

请求处理

每个块设备驱动程序的核心是它的请求函数。实际的工作，至少如设备的启动，都是在这个函数里完成的。因此我们将花很长的篇幅讨论块设备驱动程序中的请求处理过程。

磁盘驱动程序的性能，是整个操作系统性能的重要组成部分。因此内核的块设备子系统在编写的时候就非常注意性能方面的问题，除了从所控制的设备上获得信息以外，块设备子系统为驱动程序完成了所有可能的工作。这是非常有益的，因为它使得快速I/O成为可能。从另外一个角度讲，块设备子系统不必关心驱动程序API的大量复杂性。编写一个非常简单的*request*函数（很快读者将要看到）是可行的，但是如果驱动程序要在很高的层次上控制非常复杂的硬件，那么它将不再简单。

request函数介绍

块设备驱动程序的*request*函数有以下原型：

```
void request(request_queue_t *queue);
```

当内核需要驱动程序处理读取、写入以及其他对设备的操作时，就会调用该函数。在其返回前，*request*函数不必完成所有在队列中的请求；事实上，对大多数真实设备而言，它可能没有完成任何请求。然而它必须启动对请求的响应，并且保证所有的请求最终被驱动程序所处理。

每个设备都有一个请求队列。这是因为对磁盘数据实际的传入和传出发生的时间，与内核请求的时间相差很大，因此内核需要有一定的灵活性，以安排在适当时刻（比如把影响相邻磁盘扇区的请求分成一组）进行传输。还有请记住，当请求队列生成的时候，*request*函数就与该队列绑定在一起。现在回过头来看看*sbull*是如何处理它的队列的：

```
dev->queue = blk_init_queue(sbull_request, &dev->lock);
```

当创建队列时，*request*函数绑定了它，并且提供了一个自旋锁做为队列创建过程的一部分。当调用*request*函数时，该锁是由内核控制的。因此*request*函数是一个原子上下文中运行的；它必须遵从在第五章所讲的原子操作代码的一般规则。

当*request*函数拥有自旋锁时，该锁防止内核为设备安排其他请求。在一些情况下，可能需要当*request*运行的时候解锁。如果要这样做，当锁打开的时候，必须保证禁止对请求队列，或者是其他被该锁保护的数据结构的访问。在*request*函数返回前，必须重新获得该锁。

最后，对*request*函数的调用（通常）是与用户空间进程中的动作完全异步的。我们不

能假设内核正运行在初始化当前请求进程的上下文中。我们也无法知道请求所提供的I/O缓存是在内核空间中还是在用户空间中。因此直接对用户空间的任何类型的访问都会导致错误。正如读者看到的，驱动程序所需要知道的任何关于请求的信息，都包含在通过请求队列传递给我们的结构中。

一个简单的request函数

*sbull*例子驱动程序为请求处理提供了许多不同的方法。在默认的情况下，*sbull*使用名为*sbull_request*的函数，它可能是一个最简单的*request*函数。下面是它的代码：

```
static void sbull_request(request_queue_t *q)
{
    struct request *req;

    while ((req = elv_next_request(q)) != NULL) {
        struct sbull_dev *dev = req->rq_disk->private_data;
        if (! blk_fs_request(req)) {
            printk (KERN_NOTICE "Skip non-fs request\n");
            end_request(req, 0);
            continue;
        }
        sbull_transfer(dev, req->sector, req->current_nr_sectors,
                req->buffer, rq_data_dir(req));
        end_request(req, 1);
    }
}
```

该函数使用了request结构。在以后的部分将仔细讲解request结构。现在读者只要认为它代表了一个块设备的I/O执行请求就可以了。

内核提供了函数*elv_next_request*用来获得队列中第一个未完成的请求；当没有请求需要处理时，该函数返回NULL。请注意*elv_next_request*并不从队列中删除请求。如果不加以干涉而两次调用该函数，则两次都返回相同的request结构。在这个简单的操作中，只有当请求完成后，它们才离开队列。

一个块请求队列可以包含那些实际并不向磁盘读出写入数据的请求。这些请求包括生产商信息，底层诊断操作，或者是与特殊设备模式相关的指令，比如对可记录介质的写模式的设定。大多数块设备不知道如何处理这些请求，只是让这些请求失败而已；*sbull*也是按照这种方式运行的。*block_fs_request*调用告诉用户该请求是否是一个文件系统请求——移动块数据的请求。如果一个请求不是文件系统请求，则将其传递给*end_request*：

```
void end_request(struct request *req, int succeeded);
```

当处理非文件系统请求时，传递0表示不能成功地完成该请求。否则调用*sbull_transfer*对数据进行实际上的移动。该函数使用了request结构中的诸多成员：

`sector_t sector;`

在设备上开始扇区的索引号。请记住这个扇区号与所有在内核与驱动程序间传递的类似序号一样，指的是512字节的扇区。如果硬件使用不同的扇区大小，需要对其按比例缩放。比如硬件使用的2048字节的扇区，在把其放入请求中时，要将开始扇区号除以4。

`unsigned long nr_sectors;`

需要传输的扇区（512字节）数。

`char *buffer;`

要传输或者要接收数据的缓冲区指针。该指针在内核虚拟地址中，如果有需要，驱动程序可以直接引用它。

`rq_data_dir(struct request *req);`

这个宏从request中得到传输的方向。返回值为0表示从设备读数据，非0表示向设备写数据。

有了这些信息，*sbull*驱动程序可以简单的使用*memcpy*调用来完成实际的数据传输。完成拷贝操作的函数（*sbull_transfer*）还处理和缩放扇区的大小，以保证拷贝操作不会超出虚拟设备的范围：

```
static void sbull_transfer(struct sbull_dev *dev, unsigned long sector,
        unsigned long nsect, char *buffer, int write)
{
    unsigned long offset = sector*KERNEL_SECTOR_SIZE;
    unsigned long nbytes = nsect*KERNEL_SECTOR_SIZE;

    if ((offset + nbytes) > dev->size) {
        printk (KERN_NOTICE "Beyond-end write (%ld %ld)\n", offset, nbytes);
        return;
    }
    if (write)
        memcpy(dev->data + offset, buffer, nbytes);
    else
        memcpy(buffer, dev->data + offset, nbytes);
}
```

使用上面的代码，*sbull*实现了完全的、简单的、基于RAM的磁盘设备。然而对许多类型的设备来说，出于多种原因，它并不是一个实际的驱动程序。

第一个原因是*sbull*是同步执行请求的，每次一条。高性能的磁盘设备具有同时处理多条请求的能力；在主板上的磁盘控制器能以最优的顺序执行它们（希望是这样）。只要处

理了队列中的第一条请求，那么在同一时间内队列中就不能有多个请求了。为了能对多于一条的请求进行处理，就需要对请求队列以及 request 结构有更深入的了解，后面的一些章节将帮助读者达到这个目的。

还有一个问题需要考虑。当系统对磁盘中连续的扇区进行大数据量的读写时，我们能够获得磁盘设备最好的性能。对磁盘操作的最昂贵的代价总是确定读写数据开始的位置；一旦位置确定了后，实际用于读取或者写入数据的时间几乎是无关紧要的。设计和实现文件系统和虚拟内存的开发人员深谙此点，因此他们尽量在磁盘上连续放置相关数据，这样在一个请求中就能传输尽可能多的扇区。块设备子系统也是这样做的；请求队列包含了大量逻辑，用来发现相邻请求并把它们集合成一个更大的操作。

*sbull*驱动程序没有做更多的上述工作，只是简单地忽略了它们。一次只能传输一个缓存的内容，这意味着一次最大的传输量几乎不能超越单页的大小。一个块设备驱动程序可以做得比这更好，但是这需要对 request 结构和建立请求的 bio 结构有更深的了解。

下面几个小节将对块设备层如何完成其工作，以及所创建的数据结构做更深入的介绍。

请求队列

从简单的直觉上讲，一个块设备请求队列可以这样描述：包含块设备I/O请求的序列。如果只是这样理解，一个请求队列将产生极其复杂的数据结构。幸运的是，驱动程序基本不需要考虑这种复杂性。

请求队列跟踪未完成的块设备的I/O请求。但是它们在创建这些请求时也充当了一个苛刻的角色。请求队列保存了描述设备所能处理的请求的参数：最大尺寸、在同一个请求中所能包含的独立段的数目、硬件扇区的大小、对齐需求等等。如果请求队列被合理的配置，就不会提供设备不能处理的请求。

请求队列还实现了插件接口，使得多个I/O调度器的使用成为可能。一个I/O调度器的作用是以性能最大化为目的，为驱动程序提供I/O请求。大多数I/O调度器积累了大量的请求，根据块索引号升序（或者降序）排列它们，并按照这种顺序向驱动程序发送请求。当给出一个排列好的请求列表后，磁头将从一个磁盘的末尾移向另外一个磁盘，如同单向电梯一样，直到每个请求（等待出电梯的人们）都得到满足。2.6版本内核包含了一个“最终期限调度器”，它努力使得每个请求在预先设定的最长时间内得到满足；还包含了一个“预计调度器”，它实际上在一个读操作后暂时延缓了设备以期望另外一个读操作的到来。在编写本书的时候，默认的调度器是“预计调度器”，它似乎能提供最好的系统交互性能。

I/O调度器还负责合并邻近的请求。当新的I/O请求被传递给调度器时，它将在队列中搜索包含邻近扇区的请求；如果发现一个这样的请求，并且该请求并不费时，则合并这两个请求。

请求队列拥有request_queue或者request_queue_t结构类型。在<*linux/blkdev.h*>中定义了该结构及操作该结构的函数。如果对请求队列的实现感兴趣，可以在*drivers/block/ll_rw_block.c*和*elevator.c*中找到大部分代码。

队列的创建与删除

在例子代码中可以看到，一个请求队列就是一个动态的数据结构，该结构必须由块设备的I/O子系统创建。创建和初始化请求队列的函数是：

```
request_queue_t *blk_init_queue(request_fn_proc *request, spinlock_t *lock);
```

该函数的参数是处理这个队列的*request*函数指针和控制访问队列权限的自旋锁。由于该函数负责分配内存（实际上是相当多的内存），因此可能会失败；所以在使用队列前一定要检查返回值。

作为初始化请求队列的一部分，可以把成员queuedata（是一个void类型的指针）设置为任意值。该成员在请求队列中的作用与已经探讨过的其他数据结构中的private_data是等同的。

为了把请求队列返回给系统（通常在模块卸载的时候），需要调用*blk_cleanup_queue*：

```
void blk_cleanup_queue(request_queue_t *);
```

调用该函数后，驱动程序将不会再得到这个队列中的请求，也不能再引用这个队列了。

队列函数

只有很少的几个函数负责处理队列中的请求 —— 至少对驱动程序来说是这样的。在调用这些函数前，必须拥有队列锁。

返回队列中下一个要处理的请求的函数是*elv_next_request*：

```
struct request *elv_next_request(request_queue_t *queue);
```

在*sbull*例子程序中，已经看到过这个函数了。它返回下一个需要处理的请求指针（由I/O调度器决定），如果没有请求需要处理，则返回NULL。*elv_next_request*依然将请求保存在队列中，但是为其做了活动标记；该标记保证了当开始执行该请求时，I/O调度器不再将该请求与其他请求合并。

将请求从队列中实际删除，使用 *blkdev_dequeue_request* 函数：

```
void blkdev_dequeue_request(struct request *req);
```

如果驱动程序同时处理同一队列中的多个请求，则驱动程序必须按此方式将它们拿出队列。

如果出于某些原因要将拿出队列的请求再返回给队列，使用下面的函数：

```
void elv_requeue_request(request_queue_t *queue, struct request *req);
```

队列控制函数

驱动程序可以使用块设备层导出的一组函数去控制请求队列的操作。这些函数包括：

```
void blk_stop_queue(request_queue_t *queue);
void blk_start_queue(request_queue_t *queue);
```

如果驱动程序进入不能再处理更多命令的状态，它将调用*blk_stop_queue*以通知块设备层。调用该函数后，*request* 函数在驱动程序调用 *blk_start_queue* 之前，不会再被调用了。顺理成章的是，当驱动程序能够处理更多请求时，不要忘记重新开始队列。当调用上述两个函数时，队列锁应该被锁住。

```
void blk_queue_bounce_limit(request_queue_t *queue, u64 dma_addr);
```

该函数告诉内核驱动程序执行DMA所使用的最高物理内存。如果一个请求包含了超越界限的内存引用，将使用回弹缓冲区（bounce buffer）进行处理。当然如此执行块设备I/O的代价是高昂的，因此要尽量避免。可以把任何可行的物理地址作为该函数的参数，或者使用预定义好的BLK_BOUNCE_HIGH（对高端内存页使用回弹缓冲区）、BLK_BOUNCE_ISA（驱动程序只能在16MB的ISA区执行DMA）、BLK_BOUNCE_ANY（驱动程序能在任何地址执行DMA）。默认值是BLK_BOUNCE_HIGH。

```
void blk_queue_max_sectors(request_queue_t *queue, unsigned short max);
void blk_queue_max_phys_segments(request_queue_t *queue, unsigned short max);
void blk_queue_max_hw_segments(request_queue_t *queue, unsigned short max);
void blk_queue_max_segment_size(request_queue_t *queue, unsigned int max);
```

这些函数负责设置描述请求的参数，以满足驱动程序的需要。*blk_queue_max_sectors* 用（512字节）扇区为单位设置请求的最大值，默认值是255。*blk_queue_max_phys_segments* 和 *blk_queue_max_hw_segments* 用来控制在一个请求中包含了多少个物理段（在系统内存中的非连续区成员）。*blk_queue_max_phys_segments*表示驱动程序准备处理多少个段；它可能是一个静态分配的分

散表的大小。与此相反，*blk_queue_max_hw_segments*指的是驱动程序自身可以处理的段的最大数量。这两个参数的默认值都是128。最后*blk_queue_max_segment_size*以字节为单位告诉内核一个请求中单独段的大小，其默认值是65536字节。

`blk_queue_segment_boundary(request_queue_t *queue, unsigned long mask);`
一些设备无法处理那些跨越特定大小内存边界的请求；如果用户使用的是这样的设备，使用该函数告诉内核特定的边界。比如设备不能处理跨越4MB边界的请求，则mask的值为`0x3fffff`。默认的mask是`0xffffffff`。

`void blk_queue_dma_alignment(request_queue_t *queue, int mask);`
该函数告诉内核设备在使用DMA传输时的内存对齐限制。所有请求都是按照指定的对齐方式创建的，并且请求的大小也与对齐方式所匹配。默认的mask是`0x1ff`，它使得所有的请求都使用512字节对齐方式。

`void blk_queue_hardsect_size(request_queue_t *queue, unsigned short max);`
该函数告诉内核设备硬件的扇区大小。所有由内核生成的请求都是该大小的整数倍，并且做到了边界对齐。在块设备层和驱动程序之间的通信都是以512字节的扇区为单位的。

请求过程剖析

在例子程序中，使用了`request`结构。但对这个复杂的数据结构也仅仅是浅尝即止。在本节中，将对其进行仔细的分析，并了解块设备I/O在Linux内核中是如何表示的。

每个`request`结构都代表了一个块设备的I/O请求，在较高的层次，它可能是通过对多个独立请求合并而来。为了一个特定的请求而传输的扇区可能分布在整个内存中，但是通常在块设备中，它们是多个连续的扇区。请求被表示为一系列段，每个段都对应内存中的一个缓冲区。如果多个请求都是对磁盘中相邻扇区进行操作，则内核将合并它们，但是内核不会合并在单独`request`结构中的读写操作。如果合并的结果会打破对请求队列的限制，则内核也不会对请求进行合并。

从本质上讲，一个`request`结构是作为一个`bio`结构的链表实现的。当然是依靠一些管理信息来组合的，这样保证在执行请求的时候，驱动程序能知道执行的位置。`bio`结构是在底层对部分块设备I/O请求的描述。下面对其进行详细描述。

bio结构

当内核以文件系统、虚拟内存子系统或者系统调用的形式决定从块I/O设备输入、输出

块数据时，它将再结合一个bio结构，用来描述这个操作。该结构被传递给I/O代码，代码会把它合并到一个已经存在的request结构中，或者根据需要，再创建一个新的request结构。bio结构包含了驱动程序执行请求的全部信息，而不必与初始化这个请求的用户空间的进程相关联。

bio结构在<*linux/bio.h*>中定义，其中包含了驱动程序作者所要使用的诸多成员。

`sector_t bi_sector;`

该bio结构所要传输的第一个（512字节）扇区。

`unsigned int bi_size;`

以字节为单位所需传输的数据大小。通常使用`bio_sectors(bio)`宏获得每个扇区的大小。

`unsigned long bi_flags;`

bio中一系列的标志位；如果是写请求，最低有效位将被设置（使用`bio_data_dir(bio)`宏，而不是直接查看该标志）。

`unsigned short bio_phys_segments;`

`unsigned short bio_hw_segments;`

当DMA映射完成时，它们分别表示BIO中包含的物理段的数目和硬件所能操作的段的数目。

bio结构的核心是一个名为bi_io_vec的数组，它是由下面的结构组成的：

```
struct bio_vec {
        struct page     *bv_page;
        unsigned int    bv_len;
        unsigned int    bv_offset;
};
```

图16-1显示了这些结构是如何紧密结合的。正如读者看到的，当块设备I/O请求被转换到bio结构后，它将被单独的物理内存页所销毁。驱动程序所做的所有工作就是根据这个结构数组（bi_vcnt构成），使用每页传输数据（在偏移位置传输len个字节）。

为了让内核开发者能在未来修改bio结构，而又不用重新编写驱动程序代码，并不推荐直接使用bi_io_vec。因此在使用bio结构的时候，提供了一套宏来简化工作过程。在使用bio来操作每个段的开始阶段，它只是简单地在bi_io_vec数组中遍历每个没有被处理的入口。使用下面的宏完成这个工作：

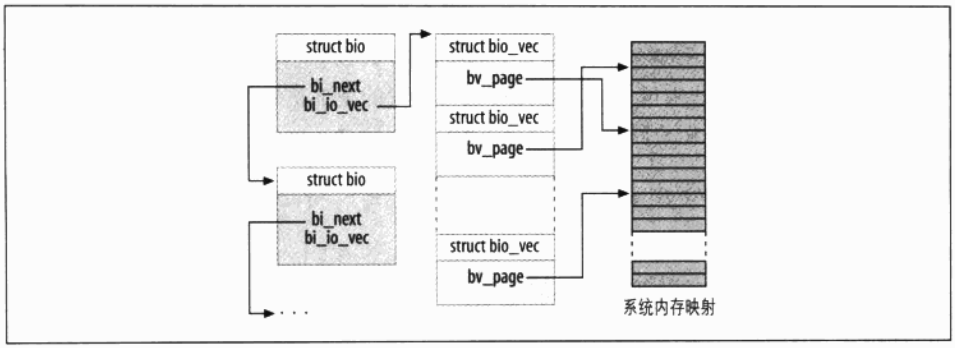


图 16-1: bio 结构

```
int segno;
struct bio_vec *bvec;

bio_for_each_segment(bvec, bio, segno) {
    /* 使用该段进行一定的操作 */
}
```

在遍历循环过程中，bvec 指向当前的 bio_vec 入口，segno 是当前的段号。使用这些值来建立DMA传输（另外一种使用*blk_rq_map_sg*的方法将在“块设备请求和DMA”一节中讲述）。如果需要直接访问页，需要首先保证正确的内核虚拟地址是存在的。为达到这个目的，可以使用：

```
char *__bio_kmap_atomic(struct bio *bio, int i, enum km_type type);
void __bio_kunmap_atomic(char *buffer, enum km_type type);
```

这个底层函数直接映射了指定索引号为 i 的 bio_vec 中的缓冲区。一个原子kmap被创建；调用者必须提供正确的槽以供使用（在第十五章中的“内存映射和struct page结构”中讲述）。

为了能追踪正在处理的请求的当前状态，块设备层也在 bio 结构中维护了一系列指针。提供了一些宏用来访问这个状态：

`struct page *bio_page(struct bio *bio);`

返回指向下一个传输页的 page 结构的指针。

`int bio_offset(struct bio *bio);`

返回在页中被传输数据的偏移量。

`int bio_cur_sectors(struct bio *bio);`

返回要在当前页中传输的扇区数量。

```
char *bio_data(struct bio *bio);
```

返回指向被传输数据的内核逻辑地址。请注意只有当正在处理的页不在高端内存时，该地址才有效。在默认的情况下，块设备子系统不会把高端内存中的缓冲区传递给驱动程序，但是，如果使用*blk_queue_bounce_limit*改变了这一设置，就不应该再使用bio_data了。

```
char *bio_kmap_irq(struct bio *bio, unsigned long *flags);
void bio_kunmap_irq(char *buffer, unsigned long *flags);
```

*bio_kmap_irq*可以为任何缓冲区返回内核虚拟地址，而不管其是在高端内存还是在低端内存中。使用了一个原子kmap，因此驱动程序在这个映射处于活动状态时不能睡眠。使用*bio_kunmap_irq*可取消缓冲区映射。请注意这里的flags参数是以指针形式传递的。还要注意的是，由于使用了原子kmap，因此在同一时刻，不能映射超过一个段。

刚才讲述的所有函数在访问“当前”缓冲区——第一个缓冲区时，直到内核知道前，缓冲区还没有被传输。驱动程序经常希望在通知完成任何缓冲区（使用*end_that_request_first*，一会将要讲到）前，完成bio中的多个缓冲区，因此这些函数的用处就不大了。还有一些其他的宏用于bio结构的内部（参看<*linux/bio.h*>了解详细情况）。

request结构成员

我们已经知道bio结构是如何工作的了，现在要仔细探究request结构，以了解请求过程的工作情况。该结构中的成员包括：

```
sector_t hard_sector;
unsigned long hard_nr_sectors;
unsigned int hard_cur_sectors;
```

用于追踪那些驱动程序还未完成的扇区。还未传输的第一个扇区保存在hard_sector中，等待传输扇区的总数量保存在hard_nr_sectors中，当前bio中剩余的扇区数目包含在hard_cur_sectors中。这些成员只能被块设备子系统所使用，驱动程序不能使用它们。

```
struct bio *bio;
```

bio是该请求的bio结构链表。不能直接对该成员进行访问；而要使用*rq_for_each_bio*（后面讲述）访问。

`char *buffer;`

本章前面那个简单的例子中，使用该成员来查找需要传输的缓冲区。随着理解的不断深入，可以看到这个成员不过是在当前 bio 中调用 *bio_data* 的结果。

`unsigned short nr_phys_segments;`

该值表示当相邻的页被合并后，在物理内存中被这个请求所占用的段的数目。

`struct list_head queuelist;`

它是一个内核链表结构（在第十一章的“链表”一节中讲述），用来把请求链接到请求队列中。如果（只有当）使用 *blkdev_dequeue_request* 函数把请求从队列中删除时，我们可以使用这个链表来跟踪由驱动程序维护的内部链表中的请求。

图 16-2 显示了 request 结构和它的 bio 结构是如何结合在一起的。在图中，请求被部分处理了，而 `cbio` 和 `buffer` 成员指向第一个未被传输的 bio 结构。

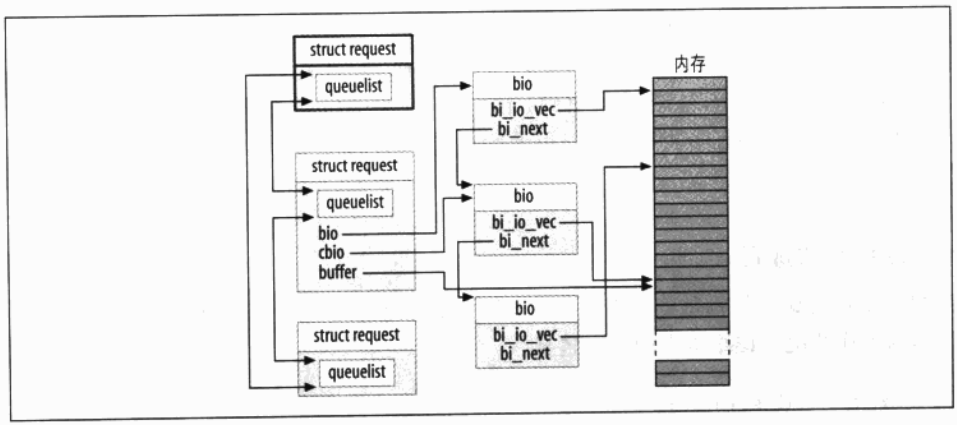


图 16-2：有一个正在处理请求的请求队列

在request结构中，还有许多其他的成员，但是本节中的列表能满足大部分驱动程序作者的需要。

屏障请求

在驱动程序接收到请求前，块设备层重新组合了请求以提高I/O性能。出于同样的目的，驱动程序也可以重新组合请求。通常重新组合是发生在将多个请求发送给驱动程序并让硬件解决优化顺序时。但是在无限制重新组合请求时面临了一个问题：一些应用程序的某些操作，要写在另外一些操作开始前完成。比如关系数据库就必须保证在执行一个关于数据库内容的会话前，日志信息要写到驱动器上。日志文件系统目前已经在大多数

Linux 系统中使用，它就有非常相似的限制。如果重新组合了错误的操作，将会导致严重的、无法检测的数据破坏。

2.6 版本的块设备层使用屏障（barrier）请求来解决这个问题。如果一个请求被设置了 REQ_HARDBARRER 标志，那么在其他后续请求被初始化前，它必须被写入驱动器。“写入驱动器”的意思是数据必须持久保存在物理介质中。许多驱动器会缓存写请求；这种缓存提高了性能，但是也违背了屏障请求的初衷。如果当重要数据还在驱动器的缓存中时发生了掉电故障，即使驱动器报告没发生错误，数据也会丢失。因此一个实现屏障请求的驱动程序一定要采取措施，使得驱动器将数据真正写入介质中。

如果驱动程序要实现屏障请求，所要做的第一步是将这一特性通知块设备层。屏障操作是另外一个请求队列；用下面的函数设置：

```
void blk_queue_ordered(request_queue_t *queue, int flag);
```

指出驱动程序要实现屏障请求，将 flag 参数设置为非零值。

屏障请求的实现只是检测 request 结构中的相关标志，为此，内核提供了一个宏完成这个工作：

```
int blk_barrier_rq(struct request *req);
```

如果这个宏返回一个非零值，该请求是一个屏障请求。由硬件的工作方式所决定，可以在一个屏障请求完成前，停止从队列中取请求。另外一些驱动器自己就能处理屏障请求，在这种情况下，驱动程序所要做的只是为这些驱动器提供正确的操作。

不可重试请求

当第一次请求失败后，块设备驱动程序经常要重试请求。这样的性能使得系统更可靠，不会丢失数据。然而内核有些时候标记请求是不可重试的。这些请求如果在第一次执行失败后，要尽快抛弃。

如果驱动程序要重试一个失败的请求，首先它要调用：

```
int blk_noretry_request(struct request *req);
```

如果该宏返回一个非零值，那么驱动程序只要忽略这个请求并返回错误值就可以了，不用重试它。

请求完成函数

正如已经看到的，还有许多不同方法处理 request 结构。它们都使用一系列的常用函

数，来处理全部的I/O请求，或者请求的一部分。所有这些函数都是原子操作的，因此可以从原子上下文中被安全调用。

当设备完成在一个I/O请求的部分或者全部的扇区时，它必须调用下面的函数通知块设备子系统：

```
int end_that_request_first(struct request *req, int success, int count);
```

该函数告诉块设备代码：驱动程序从前一次结束的地方开始，完成了规定数目的扇区的传输。如果I/O成功，则传递1表示成功；否则传递0。请注意必须报告从第一个扇区到最后一个扇区的完成情况；如果驱动程序和设备因不明原因颠倒完成请求的顺序，必须保存颠倒完成的状态直到涉及的扇区传输完毕。

*end_that_request_first*的返回值表明该请求中的所有扇区是否被传输。如果返回值是0表示所有的扇区都被传输了，该请求执行完毕。此时必须使用*blkdev_dequeue_request*函数删除请求（如果还没有做这步），并把其传递给：

```
void end_that_request_last(struct request *req);
```

*end_that_request_last*通知任何等待已经完成请求的对象，并重复利用该request结构；调用该函数时，必须拥有队列锁。

在我们的简单*sbull*例子程序中，并未使用上面所述的任何函数。使用这些函数的例子是下面的*end_request*函数。为了显示该函数的效果，下面是在2.6.10内核中*end_request*函数的全部代码：

```
void end_request(struct request *req, int uptodate)
{
    if (!end_that_request_first(req, uptodate, req->hard_cur_sectors)) {
        add_disk_randomness(req->rq_disk);
        blkdev_dequeue_request(req);
        end_that_request_last(req);
    }
}
```

*add_disk_randomness*函数使用块设备I/O请求时间来增加系统的随机数池熵。只有当磁盘传输速率确实是随机的时候才使用该函数。对于大多数机械设备来说是这样的，但是对于基于内存的虚拟设备来说就不同了，比如*sbull*。因此在下一节中复杂版本的*sbull*将不使用*add_disk_randomness*函数。

使用bio

现在读者有足够的知识通过直接操作组成请求的bio结构来编写一个块设备驱动程序

了。参看例子代码是很有帮助的。如果request_mode参数设置为1时加载*sbull*驱动程序，它将注册一个bio请求函数，来代替前面看到的简单函数。该函数如下：

```
static void sbull_full_request(request_queue_t *q)
{
    struct request *req;
    int sectors_xferred;
    struct sbull_dev *dev = q->queuedata;

    while ((req = elv_next_request(q)) != NULL) {
        if (! blk_fs_request(req)) {
            printk (KERN_NOTICE "Skip non-fs request\n");
            end_request(req, 0);
            continue;
        }
        sectors_xferred = sbull_xfer_request(dev, req);
        if (! end_that_request_first(req, 1, sectors_xferred)) {
            blkdev_dequeue_request(req);
            end_that_request_last(req);
        }
    }
}
```

该函数只是获得了每一个请求，并把它们传递给*sbull_xfer_request*，然后使用*end_that_request_first*完成请求，如果有必要的话，还使用*end_that_request_last*函数。因此该函数可以处理部分高级队列和请求管理问题。实际执行一个请求的工作交给*sbull_xfer_request*函数完成：

```
static int sbull_xfer_request(struct sbull_dev *dev, struct request *req)
{
    struct bio *bio;
    int nsect = 0;

    rq_for_each_bio(bio, req) {
        sbull_xfer_bio(dev, bio);
        nsect += bio->bi_size/KERNEL_SECTOR_SIZE;
    }
    return nsect;
}
```

这里介绍另外一个宏：*rq_for_each_bio*。正如读者想像的那样，该宏只是遍历了请求中的每个bio结构，并且提供了可以传递给*sbull_xfer_bio*函数用于传输的指针。下面是该函数的代码：

```
static int sbull_xfer_bio(struct sbull_dev *dev, struct bio *bio)
{
    int i;
    struct bio_vec *bvec;
    sector_t sector = bio->bi_sector;
```

```
        /* 对每个段独立操作 */
        bio_for_each_segment(bvec, bio, i) {
            char *buffer = __bio_kmap_atomic(bio, i, KM_USER0);
            sbull_transfer(dev, sector, bio_cur_sectors(bio),
                    buffer, bio_data_dir(bio) ==  = WRITE);
            sector += bio_cur_sectors(bio);
            __bio_kunmap_atomic(bio, KM_USER0);
        }
        return 0;
        /* 总是"成功" */
    }
```

该函数遍历了bio结构中的每个段，获得内核虚拟地址以访问缓冲区，然后调用前面介绍过的*sbull_transfer*函数，以完成数据的拷贝。

每个设备都有自己的需求，但是作为一条通用的准则，上述代码为众多情况提供了一种模型，在这种模型中，充分利用bio结构是必要的。

块设备请求和DMA

如果使用的是一个高性能块驱动程序，在实际传输过程中可以使用DMA。一个块设备驱动程序可以像前面讨论的那样遍历bio结构，并为每个bio结构创建DMA映射，并将结果传递给设备。如果读者的设备可以完成"分散/聚集"I/O，有一个更简便的实现方法。函数：

```
int blk_rq_map_sg(request_queue_t *queue, struct request *req,
                  struct scatterlist *list);
```

从指定的请求中获得全部的段，然后把它们填写到给定的表中。在内存中相邻的段将被结合在分散表插入点的前面，因此无需检测它们。返回值表示的是表中入口项的个数。该函数会使用第三个参数返回一个分散表，它可用来传递给*dma_map_sg*（请参看第十五章"分散－聚集映射"获得关于*dma_map_sg*的知识）。

在调用*blk_rq_map_sg*前，驱动程序必须为分散表分配存储空间。该表必须能容纳至少与请求所拥有的物理段同样多的入口项。request结构中nr_phys_segments成员包含了这个数量，它不能超过*blk_queue_max_phys_segments*所设定的物理段的最大值。

如果不想让*blk_rq_map_sg*合并相邻的段，可以使用下面的函数改变这个默认的行为：

```
clear_bit(QUEUE_FLAG_CLUSTER, &queue->queue_flags);
```

一些SCSI磁盘驱动程序使用这个方法标志其请求队列，因为它们不能从合并请求中获得更好的性能。

不使用请求队列

前面提到内核会优化队列中的请求顺序，这包括排列请求，甚至可能要停止某些请求，以期望某些其他请求的到来。在处理实际的旋转磁盘驱动器的时候，这能帮助系统获得最好的性能。但是在处理类似*sbull*这样的设备时，这些优化过程有些浪费。许多面向块数据的设备，比如flash内存阵列、数字相机使用的读卡器、RAM盘，它们都是完全的随机访问设备，并不能从高级请求队列逻辑中获益。另外一些设备，比如软件RAID组，或者是逻辑卷标管理器创建的虚拟磁盘，并不具备块设备层请求队列优化所需要的性能特点。对于这些设备，最好还是从块设备层中直接接受请求，而不要去打乱请求队列的好。

在这些情况下，块设备层支持“无队列”模式的操作。为了能使用该模式，驱动程序必须提供一个“构造请求”的函数，而不是一个请求处理函数。下面是构造请求函数的原型：

```
typedef int (make_request_fn) (request_queue_t *q, struct bio *bio);
```

请注意：虽然从不拥有一个请求，但是还依然提供了一个请求队列。*make_request*函数的主要参数是bio结构，它表示了要被传输的一个或者多个缓冲区。*make_request*函数能够完成下面的事情：直接进行传输，或者把请求重定向给其他设备。

直接进行传输是利用前面介绍过的方法，通过bio进行传输。由于没有request结构进行操作，因此函数应该能够调用*bio_endio*，告诉bio结构的创建者请求的完成情况：

```
void bio_endio(struct bio *bio, unsigned int bytes, int error);
```

这里bytes是所要传输的字节数。它可以比bio结构中表示的字节数小；此时可以通知“部分完成”，然后更新bio结构中的“当前缓冲区”指针。当设备需要再次传输时，可以再次调用*bio_endio*，如果不能完成这个请求，则给出一个错误。通过给error参数赋予一个非零的数来表示错误；这个值通常是诸如-EIO这样的错误码。无论I/O成功与否，*make_request*都返回0。

如果使用request_mode为2来加载*sbull*，它将使用*make_request*函数。由于*sbull*已经有一个能传输单独bio的函数了，因此*make_request*函数很简单：

```
static int sbull_make_request(request_queue_t *q, struct bio *bio)
{
    struct sbull_dev *dev = q->queuedata;
    int status;

    status = sbull_xfer_bio(dev, bio);
    bio_endio(bio, bio->bi_size, status);
    return 0;
}
```

请注意绝对不要在普通的 *request* 函数中调用 *bio_endio*；而要调用 *end_that_request_first* 函数。

一些块设备驱动程序，比如那些实现了卷标管理器和软件RAID组的驱动程序，实际上需要将请求重定向给其他能处理该请求的设备。编写这样的程序已经超越了本书的范围。这里提醒读者，如果 *make_request* 函数返回了一个非零的值，bio将再次被提交。一个“堆积”驱动程序能够修改bi_bdev成员，以指向不同的设备，修改开始扇区的值，然后返回。块设备系统接着将bio传给新设备。还有一个 *bio_split* 函数，为了把bio提交给多个设备，这个函数可将bio分成若干个的小块。如果queue参数设置正确，则以这样的方式分割bio几乎没有必要。

另外还必须告诉块设备子系统，驱动程序使用定制的 *make_request* 函数。为做到这点，必须使用下面的函数分配一个请求队列：

```
request_queue_t *blk_alloc_queue(int flags);
```

该函数与 *blk_init_queue* 的不同在于它并未真正的建立一个保存请求的队列。flags参数是一系列分配标志，用来为队列分配内存；通常正确的值是GFP_KERNEL。一旦拥有了队列，将它与 *make_request* 函数传递给 *blk_queue_make_request*：

```
void blk_queue_make_request(request_queue_t *queue, make_request_fn *func);
```

sbull 代码构造 *make_request* 函数的代码如下：

```
dev->queue = blk_alloc_queue(GFP_KERNEL);
if (dev->queue =  = NULL)
    goto out_vfree;
blk_queue_make_request(dev->queue, sbull_make_request);
```

如果好奇，可以花点时间仔细看一下 *drivers/block/ll_rw_block.c*，它显示所有的队列都有 *make_request* 函数。在默认的版本中，*generic_make_request* 负责将bio合并到request结构中。通过提供自己的 *make_request* 函数，驱动程序及可以真正重载请求队列的方法，并能挑选出大部分工作。

其他一些细节

本一小节包含了关于块设备中其他方面的知识，对编写高级驱动程序会有一些帮助。编写一个正确的驱动程序，用不到下面的内容，但是在某些时候，它们是有用的。

命令预处理

块设备层为驱动程序在返回 *elv_next_request* 前，提供了检查和预处理请求的机制。该机制允许驱动程序预先建立驱动器命令，决定是否处理该请求，还是予以其他方式的处理。

如果想使用这个功能，要按照下面的原型建立命令预处理函数：

```
typedef int (prep_rq_fn) (request_queue_t *queue, struct request *req);
```

request结构包含了一个成员 —— cmd，它是一个BLK_MAX_CDB个字节的数组；预处理函数可以使用该数组保存实际的硬件命令（或者任何其他有用信息）。该函数要能返回下面的值之一：

BLKPREP_OK
: 命令预处理正常，可以发送请求到驱动程序的请求函数中。

BLKPREP_KILL
: 该请求不能被完成，失败并返回错误码。

BLKPREP_DEFER
: 目前该请求不能完成。把它放在队列的前头，但是不传递给请求函数。

在将请求返回给驱动程序前，*elv_next_request* 会立刻调用预处理函数。如果该函数返回BLKPREP_DEFER，那么*elv_next_request*返回给驱动程序的值是NULL。这种操作模式很有用，比如当设备达到其能处理的最大请求数目时。

为了让块设备层调用预处理函数，将其传递给：

```
void blk_queue_prep_rq(request_queue_t *queue, prep_rq_fn *func);
```

默认的情况下，请求队列没有预处理函数。

标记命令队列

同时拥有多个活动请求的硬件通常支持某种形式的标记命令队列（Tagged Command Queueing，TCQ）。TCQ只是为每个请求添加一个整数（标记）的技术，这样当驱动器完成它们中的一个请求后，它就可以告诉驱动程序完成的是哪个。在以前版本的内核中，实现了TCQ的块设备驱动程序自己完成所有的工作；在2.6内核中，在块设备层代码中添加了一段TCQ支持的代码，以便所有的驱动程序使用。

为了让驱动器支持标记命令队列，必须在初始化的时候调用下面的函数告诉内核：

```
int blk_queue_init_tags(request_queue_t *queue, int depth,
                        struct blk_queue_tag *tags);
```

这里，queue是请求队列，depth是在任何时刻设备所支持的未完成标记请求的个数。tag是个可选的指针，指向一个blk_queue_tag结构数组；它必须有depth个。通常传递的标记可以是NULL，*blk_queue_init_tags*负责分配数组。如果要在多个设备之间共享标记，可以从另外一个请求队列中传递标记数组指针（保存在queue_tags成员中）。绝对不要自己分配标记数组；块设备层需要初始化数组，并且不向模块提供初始化函数。

由于*blk_queue_init_tags*分配内存，因此它可能会失败。如果失败，它将返回负的错误码给调用者。

如果设备能处理的标记数量发生了变化，可以用下面的函数通知内核：

```
int blk_queue_resize_tags(request_queue_t *queue, int new_depth);
```

在调用过程中，必须锁住队列锁。该调用可能失败，若失败则返回负的错误码。

使用*blk_queue_start_tag*将一个标记与一个请求相关联，调用该函数时必须锁住队列锁：

```
int blk_queue_start_tag(request_queue_t *queue, struct request *req);
```

如果一个标记可用，则该函数为请求分配该标记，把标记编号保存在req->tag里，然后返回0。它还把请求清除出队列，并把所清除的请求连接到自己的标记跟踪结构中，因此，驱动程序一定要小心，如果正在使用标记，就不能把请求清除出队列。如果没有可用的标记，*blk_queue_start_tag*把请求放在队列中然后返回一个非零值。

当一个指定请求的全部数据传输完毕后，驱动程序使用下面的函数返回标记：

```
void blk_queue_end_tag(request_queue_t *queue, struct request *req);
```

再次强调，在调用该函数前必须锁住队列锁。当*end_that_request_first*返回0（表示请求完成），并在调用*end_that_request_last*前，调用该函数。请记住请求已经被清除出队列，因此在此时驱动程序做这件事是错误的。

如果要为指定的标记找到相应的请求（比如驱动程序报告请求完成），使用*blk_queue_find_tag*函数：

```
struct request *blk_queue_find_tag(request_queue_t *qeue, int tag);
```

返回值是相应的request结构，如果不是则表明有错误产生。

如果发生了错误，驱动程序可能不得不重新置位，或者执行其他一些操作对其控制的设备强制纠错。在这种情况下，任何未完成标记的命令都不能被执行。块设备层提供了一个函数，可以在这种情况下帮助恢复：

```
void blk_queue_invalidate_tags(request_queue_t *queue);
```

该函数返回所有的未执行的标记给缓冲池，并且把相应的请求发还给请求队列。当调用该函数时，必须锁住队列锁。

快速参考

```
#include <linux/fs.h>
int register_blkdev(unsigned int major, const char *name);
int unregister_blkdev(unsigned int major, const char *name);
```

register_blkdev 用来向内核注册一个块设备驱动程序，还可获得主设备号。一个驱动程序可以使用 *unregister_blkdev* 函数注销。

```
struct block_device_operations
```

用来保存块设备驱动程序大多数方法的数据结构。

```
#include <linux/genhd.h>
struct gendisk;
```

用来描述内核中单个块设备的结构。

```
struct gendisk *alloc_disk(int minors);
void add_disk(struct gendisk *gd);
```

用来分配 gendisk 结构并将其返回给系统的函数。

```
void set_capacity(struct gendisk *gd, sector_t sectors);
```

在 gendisk 结构中保存设备容量（用 512 字节扇区为单位）。

```
void add_disk(struct gendisk *gd);
```

向内核添加一个磁盘。一旦调用了该函数，内核就能调用磁盘方法了。

```
int check_disk_change(struct block_device *bdev);
```

用来对指定磁盘驱动器进行介质变化检查的内核函数，当介质改变被侦测到后，采取必要的清除动作。

```
#include <linux/blkdev.h>
request_queue_t blk_init_queue(request_fn_proc *request, spinlock_t *lock);
void blk_cleanup_queue(request_queue_t *);
```

用来创建和删除块设备请求队列的函数。

```
struct request *elv_next_request(request_queue_t *queue);
void end_request(struct request *req, int success);
```

*elv_next_request*获得请求队列中的下一个请求；*end_request*用在简单的驱动程序中，以完成（或部分完成）一个请求。

```
void blkdev_dequeue_request(struct request *req);
void elv_requeue_request(request_queue_t *queue, struct request *req);
```

从队列中删除一个请求的函数，如果需要，还可以把该请求放回队列。

```
void blk_stop_queue(request_queue_t *queue);
void blk_start_queue(request_queue_t *queue);
```

如果不想让自己的请求函数被调用，*blk_stop_queue*可以做到这点。为了能使请求函数被调用，必须调用*blk_start_queue*的函数。

```
void blk_queue_bounce_limit(request_queue_t *queue, u64 dma_addr);
void blk_queue_max_sectors(request_queue_t *queue, unsigned short max);
void blk_queue_max_phys_segments(request_queue_t *queue, unsigned short max);
void blk_queue_max_hw_segments(request_queue_t *queue, unsigned short max);
void blk_queue_max_segment_size(request_queue_t *queue, unsigned int max);
blk_queue_segment_boundary(request_queue_t *queue, unsigned long mask);
void blk_queue_dma_alignment(request_queue_t *queue, int mask);
void blk_queue_hardsect_size(request_queue_t *queue, unsigned short max);
```

用来设置队列参数的函数。这些参数控制了对一个特定设备请求的创建。参数的具体解释在“队列控制函数”一节。

```
#include <linux/bio.h>
struct bio;
```

表示部分块设备I/O请求的底层结构。

```
bio_sectors(struct bio *bio);
bio_data_dir(struct bio *bio);
```

这两个宏用来获得bio结构描述的大小和传输方向。

```
bio_for_each_segment(bvec, bio, segno);
```

用来遍历组成bio结构的段的伪控制结构。

```
char *__bio_kmap_atomic(struct bio *bio, int i, enum km_type type);
void __bio_kunmap_atomic(char *buffer, enum km_type type);
```

*__bio_kmap_atomic*用来为bio结构中指定的段创建内核虚拟地址。取消该映射必须使用*__bio_kunmap_atomic*。

```
struct page *bio_page(struct bio *bio);
int bio_offset(struct bio *bio);
int bio_cur_sectors(struct bio *bio);
char *bio_data(struct bio *bio);
char *bio_kmap_irq(struct bio *bio, unsigned long *flags);
void bio_kunmap_irq(char *buffer, unsigned long *flags);
```

这是一组访问宏，用来访问bio结构中的“当前”段。

```
void blk_queue_ordered(request_queue_t *queue, int flag);
int blk_barrier_rq(struct request *req);
```

如果驱动程序实现了屏障请求，则调用*blk_queue_ordered*。如果当前请求是一个屏障请求，则宏*blk_barrier_rq*返回非零值。

```
int blk_noretry_request(struct request *req);
```

该宏返回非零值表示指定的请求因错误不能再次被执行。

```
int end_that_request_first(struct request *req, int success, int count);
void end_that_request_last(struct request *req);
```

使用*end_that_request_first*表示完成一个块设备I/O请求的过程。如果该函数返回0，则表示请求已经完成，应该被传递给*end_that_request_last*。

```
rq_for_each_bio(bio, request)
```

另外一个以宏的形式实现的控制结构，它将遍历请求中的每个bio结构。

```
int blk_rq_map_sg(request_queue_t *queue, struct request *req, struct
  scatterlist *list);
```

为DMA传输，需要将缓冲区映射到指定的request中，使用这些信息填充分散表。

```
typedef int (make_request_fn) (request_queue_t *q, struct bio *bio);
```

*make_request*函数的原型。

```
void bio_endio(struct bio *bio, unsigned int bytes, int error);
```

指定的bio结构的信号完成函数。只有当驱动程序通过*make_request*函数直接从块设备层获得bio结构时，才使用该函数。

```
request_queue_t *blk_alloc_queue(int flags);
void blk_queue_make_request(request_queue_t *queue, make_request_fn *func);
```

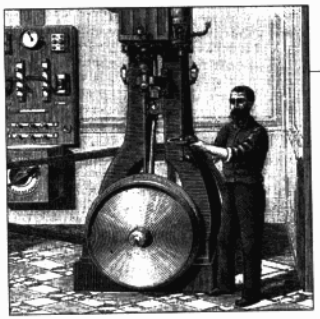
使用*blk_alloc_queue*来分配一个请求队列，以便为用户定义的*make_request*函数所使用。该函数要用*blk_queue_make_request*设置。

```
typedef int (prep_rq_fn) (request_queue_t *queue, struct request *req);
void blk_queue_prep_rq(request_queue_t *queue, prep_rq_fn *func);
```

命令预处理函数的原型和设置，它可以在请求传递到请求处理函数前，为硬件准备需要的命令。

```
int blk_queue_init_tags(request_queue_t *queue, int depth, struct
  blk_queue_tag *tags);
int blk_queue_resize_tags(request_queue_t *queue, int new_depth);
int blk_queue_start_tag(request_queue_t *queue, struct request *req);
void blk_queue_end_tag(request_queue_t *queue, struct request *req);
struct request *blk_queue_find_tag(request_queue_t *qeue, int tag);
void blk_queue_invalidate_tags(request_queue_t *queue);
```

为了让驱动程序使用标记命令队列而提供的支持函数。



第十七章

网络驱动程序

在讨论完字符设备和块设备驱动程序之后，是到进入网络世界的时候了。网络接口是第三类标准Linux设备，本章将描述网络接口是如何与内核其余的部分交互的。

系统中网络接口的角色，和一个已挂装的块设备类似。一个块设备向内核注册其磁盘和函数，然后使用其请求函数，根据请求“发送”和“接收”块数据。与此类似，网络接口也必须使用特定的内核数据结构注册自身，以备与外界进行数据包交换时调用。

在已挂装磁盘和数据包发送接口之间，还是存在着许多重要的不同之处。首先，一个磁盘在*/dev*目录下作为一个特殊文件而存在，而网络接口却没有这样的入口点。对网络接口的常用文件操作（读、写等）是没有意义的，因此在它们身上无法体现Unix的“一切都是文件”的思想。这样，网络接口存在于它们自己的名字空间中，并导出一系列不同的操作。

然而读者可能发现，当应用程序使用套接字（socket）的时候，依然使用*read*、*write*系统调用，但是这些调用作用于软件对象上，它们与网络接口完全不同。在同一个物理接口上可能存在几百个多工的套接字。

但是这两种设备之间最重要的不同是：块设备只响应来自内核的请求，而网络驱动程序异步地接收来自外部世界的数据包。因此当内核要求一个块设备驱动程序向其发送缓冲区数据时，而网络设备则向内核请求把从外部获得的数据包发送给内核。内核中的网络驱动程序接口是为不同模式的操作而精心设计的。

网络驱动程序还将准备支持大量的管理任务，比如设置地址、修改传输参数，以及维护流量和错误统计。网络驱动程序的API反映了这一需求，这使得它看起来与前面讲述的驱动接口有所不同。

Linux内核中的网络子系统被设计成完全与协议无关。该思想应用于网络协议（IP、IPX

及其他协议）和硬件协议中（以态网、令牌环等）。内核与网络驱动程序之间的交互，可能每次处理的是一个网络数据包；协议隐藏在驱动程序之后，同时物理传输又被隐藏在协议之后。

本章将讲述网络接口是如何服务于Linux内核的其余部分的，同时提供一个基于内存的模块化接口实例 —— *snull*。为了简化讨论，该接口使用了以态网硬件协议并传输IP数据包。从例子*snull*中获得的知识，可以应用于其他非IP协议，并且在非以态网驱动程序的编写中，仅在与实际网络协议相关的细节中存在微小差异。

本章不会论及IP地址编号方式、网络协议或者其他一般性的网络概念。这些内容通常不是驱动程序编写者所关心的，而用不到一百页的篇幅对网络技术进行描述，不可能达到令人满意的效果。对此感兴趣的读者可以参阅其他描述网络技术的书籍。

在讲述网络设备前，首先讲一个术语。在网络世界中使用术语“octet”指一组8个的数据位，它是能为网络设备和协议所能理解的最小单位。在本章中基本不会使用术语“字节（byte）”。为了保持使用的标准性，在讲述网络设备时使用术语“octet”。

术语“协议头（header）”也将很快被提起。协议头是在数据包中的一系列字节（错了，应该是octet），它将通过网络子系统的不同层。当一个应用程序通过TCP套接字发送一块数据时，网络子系统将把数据块分割成若干数据包，并在数据包开头加入用来描述数据流类型的TCP协议头。下层协议将在TCP协议头前接着添加IP协议头，用来指定数据包达到目的地址的路径。如果数据包通过类以态网媒介，将继续在数据包前加入以态网头，其包含的信息将由硬件来解释。网络驱动程序通常不必关心高层协议的协议头，但是它们必须负责创建硬件层的协议头。

snull设计

这一小节讨论设计*snull*网络接口的一些概念。尽管这些概念适合在页边上做注脚用，但如果不能理解它们，在使用示例代码时会遇到麻烦。

首先，也是最重要的设计决策是：示例接口仍然不依赖于任何硬件，这与本书中的示例代码相同。该限制使接口有点像回环（loopback）接口。但snull不是一个回环接口，它模拟了和远程主机的会话，从而更好的展示网络驱动程序的编写。Linux回环驱动程序其实非常简单，*drivers/net/loopback.c*就是很好的例子。

*snull*的另外一个特点是它只支持IP流。这是该接口的内部工作方式所决定的 —— 为了正确模拟一对硬件接口，*snull*必须观察并解释数据包。实际的接口不会依赖于被传输的协议，而*snull*的这一限制也不会影响本章所描述的代码片段。

分配IP号

*snull*模块创建了两个接口。这些接口与简单的回环设备不同，通过其中一个接口传输的任何数据，都将出现在另外一个接口上，而不是第一个接口本身。这就好像用户有两个外部链路，但实际上计算机只对自身做出响应。

但不幸的是，只分配一个IP号是不能实现该效果的，因为如果接口A指向的是接口B，那么内核不能通过接口A发送数据。相反内核将使用回环通道而不会通过*snull*。为了能通过*snull*接口建立通信，在传输过程中需要修改源及目的地址。换句话说，通过某一个接口发送出的数据包应该被另外一个接口接收，但是不能将外发数据包的接收者认为是本机。同样的规则也应该应用于已接收数据包的源地址。

为了实现这种“隐藏的回环”设备，*snull*接口切换源地址和目标地址的第三个octet的最低位；也就是说，它修改了C类IP号的网络编号和主机编号。其效果是，发送到网络A（连接到sn0，即第一个接口）的数据包，将在属于网络B的sn1接口上出现。

为避免涉及太多的数字，赋予相关的IP号一些符号名：

- snullnet0是连接到sn0接口的C类网络。类似地，snullnet1是连接到sn1的网络。上述网络地址仅仅在第三个octet的最低位有差别。这些网络必须拥有24位的子网掩码。
- local0是赋予sn0接口的IP地址，它属于snullnet0。和sn1关联的地址是local1。local0和local1必须在第三和第四个octet的最低位上不同。
- remote0是snullnet0网络中的一个主机，它的第四个octet和local1相同。发送到remote0的任意数据包将在接口代码修改了其网络类地址之后，到达local1。remote1属于snullnet1，它的第四个octet和local0一样。

*snull*接口的操作在图17-1中描述，其中与每个接口关联的主机名打印在该接口名的旁边。

下面是一些满足上述要求的可能网络编号。将这两行放入*/etc/networks*文件之后，就可以用名字来指代网络。这些网络编号值选择自非正式使用的IP号范围。

```
snullnet0       192.168.0.0
snullnet1       192.168.1.0
```

下面是可加入到*/etc/hosts*的可能主机IP号：

```
192.168.0.1   local0
192.168.0.2   remote0
192.168.1.2   local1
192.168.1.1   remote1
```

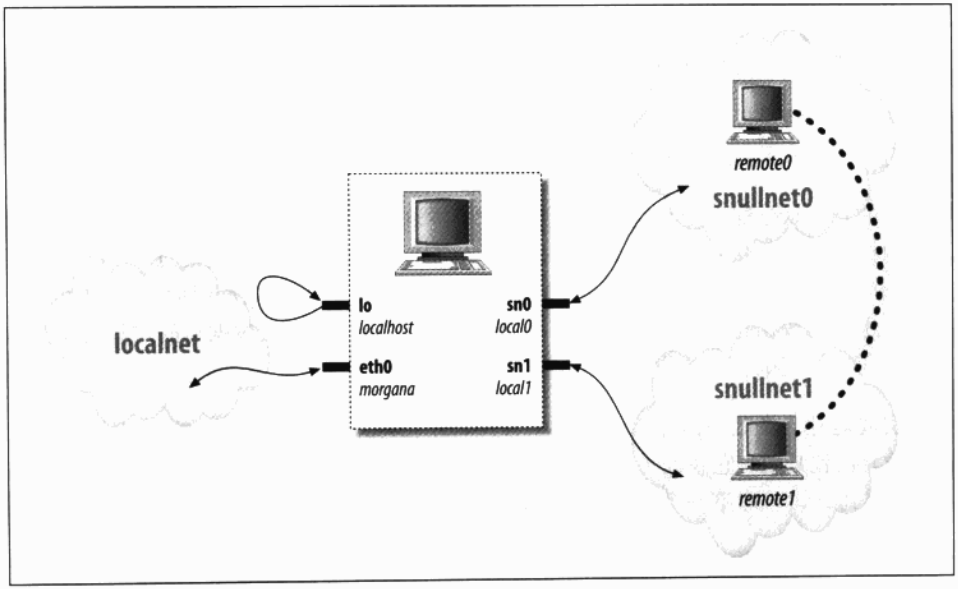


图 17-1：主机和接口的关系

上述编号一个重要特点是，local0的主机部分和remote1的主机部分一样，而local1的主机部分，和remote0的主机部分一样。只要满足上述关系，读者就可以选择一组完全不同的网络号和主机号。

如果计算机已经连入一个实际的网络，就需要小心了。用户所选择的编号可能是实际Internet或intranet的IP号，将这些编号赋予自己的接口，将导致无法和实际主机通信。例如，尽管上述编号并不是可路由的Internet编号，但可能已经在防火墙之后的内部私有网络当中使用。

不管选择什么地址编号，可通过如下命令设置接口：

```
ifconfig sn0 local0
ifconfig sn1 local1
```

如果地址选择的范围并不是C类地址，那么需要添加子网掩码255.255.255.0。

至此，就可到达接口的“远程”端。下面给出的屏幕输出说明了主机是如何通过*snull*接口到达remote0和remote1的。

```
morgana% ping -c 2 remote0
64 bytes from 192.168.0.99: icmp_seq=0 ttl=64 time=1.6 ms
64 bytes from 192.168.0.99: icmp_seq=1 ttl=64 time=0.9 ms
2 packets transmitted, 2 packets received, 0% packet loss
```

```
morgana% ping -c 2 remote1
64 bytes from 192.168.1.88: icmp_seq=0 ttl=64 time=1.8 ms
64 bytes from 192.168.1.88: icmp_seq=1 ttl=64 time=0.9 ms
2 packets transmitted, 2 packets received, 0% packet loss
```

注意，我们无法到达属于这两个网络的任意其他“主机”，这是因为当地址被修改且数据包被接收到时，计算机就会将该数据包丢弃。比如，离开sn0发送到192.168.0.32的数据包，将重新出现在sn1接口，但目标地址已修改成为192.168.1.32，这并不是主机的本地地址。

数据包的物理传输

对数据传输而言，*snull*接口属于以太网类型。

*snull*模拟以太网是因为大量已有的网络（至少一台工作站连接到的网段）基于以太网技术，比如10baseT、100baseT或者千兆以太网等等。另外，内核为以太网设备提供了一些通用支持，没有理由不利用这些通用的支持。成为一个以太网设备的优点如此出众，以至于*plip*接口（使用打印机端口的接口）也将自己声明为一个以太网设备。

对*snull*来说，使用以太网的最后一个优点是可以在该接口上运行*tcpdump*看到数据包的传输情况。利用*tcpdump*观察接口，是一种了解这两个接口工作情况的便捷途径。

先前曾提到，*snull*只能利用IP数据包。这一限制源于如下事实：为了让示例代码能正常工作，*snull*需要监听数据包，甚至修改数据包。代码要修改每个数据包IP头中的源、目标以及校验和，但不会检查数据包是否真正传送IP信息。这种“快速而恶劣的”数据修改，会破坏非IP数据包。如果读者希望通过*snull*传送其他协议，则需要修改模块源代码。

连接到内核

下面开始介绍*snull*的源代码，并开始分析网络驱动程序的结构。如果手头有若干个实际驱动程序的源代码，则能帮助读者跟上本章的讨论，并看到真实世界中，Linux网络驱动程序的工作情况。为此，推荐读者首先阅读*loopback.c*、*plip.c*和*e100.c*，这些驱动程序的复杂性是逐渐递增的。上述这些文件均保存在内核源代码树的*drivers/net*目录中。

设备注册

当一个模块被装载到正在运行的内核中时，它要请求资源并提供一些功能设施，这点上，网络驱动程序也一样，而且在资源请求的方式上也没有任何不同。驱动程序要按照第十

章“安装中断处理例程”中讲到的方法探测其设备和硬件位置（I/O端口及IRQ线），但不需要进行注册。网络驱动程序在其模块初始化函数中的注册方法，和字符驱动程序及块驱动程序不同。因为对网络接口来讲，没有和主设备号及次设备号等价的东西，所以网络驱动程序不必请求这种设备号。相反，驱动程序对每个新检测到的接口，向全局的网络设备链表中插入一个数据结构。

每个接口由一个net_device结构描述，其定义在<*linux/netdevice.h*>中。*snull*在一个数组中保存了两个指向该结构的指针（sn0和sn1）：

```
struct net_device *snull_devs[2];
```

和其他所有内核数据结构一样，net_device包含了一个kobject和引用计数，并且通过sysfs导出信息。由于与其他结构相关，因此它必须被动态分配。用来执行分配的内核函数是*alloc_netdev*，它有着如下的原型：

```
struct net_device *alloc_netdev(int sizeof_priv,
                                const char *name,
                                void (*setup)(struct net_device *));
```

这里sizeof_priv是驱动程序的“私有数据”区的大小；这个区成员和net_device结构一同分配给网络设备。实际上，它们都处于一大块内存中，但是驱动程序作者不需要知道这些。name是接口的名字，其在用户空间可见；这个名字可以使用类似*printf*中%d的格式，内核将用下一个可用的接口号替代%d。最后，setup是一个初始化函数，用来设置net_device结构剩余的部分。不久将会讨论初始化函数，但是现在已经知道*snull*用下面的代码分配它的两个设备结构：

```
snull_devs[0] = alloc_netdev(sizeof(struct snull_priv), "sn%d",
        snull_init);
snull_devs[1] = alloc_netdev(sizeof(struct snull_priv), "sn%d",
        snull_init);
if (snull_devs[0] =  = NULL || snull_devs[1] =  = NULL)
    goto out;
```

必须要检查函数的返回值，以确定分配工作成功完成。

网络子系统针对*alloc_netdev*函数，为不同种类的接口封装了许多函数。最常用的是*alloc_etherdev*，它在<*linux/etherdevice.h*>中定义：

```
struct net_device *alloc_etherdev(int sizeof_priv);
```

该函数使用eth%d的形式指定分配给网络设备的名字。它提供了自己的初始化函数（*ether_setup*），用正确的值为以态网设备设置net_device中的许多成员。因此在驱动程序中没有为*alloc_etherdev*提供初始化函数；驱动程序只是在成功分配“私有数据”

区后，直接做一些必须的初始化工作。其他类型设备的驱动程序作者也许要使用其他的封装函数，比如为光纤通道设备使用*alloc_fcdev*（在<*linux/fcdevice.h*>中定义）函数，为FDDI设备使用*alloc_fddidev*（在<*linux/fddidevice.h*>中定义）函数，或者为令牌环设备使用*alloc_trdev*（在<*linux/trdevice.h*>中定义）函数。

使用*alloc_etherdev*函数，*snull*不会遇到任何麻烦；但是这里却要使用*alloc_netdev*函数。其目的是揭示如何使用底层接口，以及如何控制分配给接口的名字。

一旦*net_device*结构被初始化后，剩余的工作就是将该结构传递给*register_netdev*函数。在*snull*中，实现该过程的代码如下：

```
for (i = 0; i < 2;  i++)
    if ((result = register_netdev(snull_devs[i])))
        printk("snull: error %i registering device \"%s\"\n",
                result, snull_devs[i]->name);
```

需要注意的是：当调用*register_netdev*函数后，就可以调用驱动程序操作设备了。因此必须在初始化一切事情后再注册。

初始化每个设备

读者已经知道了分配和注册net_device结构，但是刚才并没提到紧接着的下一步：完全初始化这个结构。请注意，在运行时，net_device结构总是被聚集在一起的；不能像处理file_operations和block_device_operations结构那样在编译时进行初始化。在调用*register_netdev*前，初始化必须完成。net_device结构既大又复杂；但幸运的是，内核在*ether_setup*函数（被*alloc_etherdev*调用）中为这个结构设置了许多默认值。

由于*snull*使用了*alloc_netdev*，因此它有独立的初始化函数。下面是该函数（*snull_init*）的核心部分：

```
ether_setup(dev); /* 对其中一些成员赋值 */

dev->open            = snull_open;
dev->stop            = snull_release;
dev->set_config      = snull_config;
dev->hard_start_xmit = snull_tx;
dev->do_ioctl        = snull_ioctl;
dev->get_stats       = snull_stats;
dev->rebuild_header  = snull_rebuild_header;
dev->hard_header     = snull_header;
dev->tx_timeout      = snull_tx_timeout;
dev->watchdog_timeo = timeout;
/* 保持默认的标志，只是添加了NOARP而已 */
dev->flags           |= IFF_NOARP;
dev->features        |= NETIF_F_NO_CSUM;
```

```
dev->hard_header_cache = NULL;
/* 禁止缓存 */
```

上面的代码是一个非常普通的初始化net_device结构的例程；它主要是用来保存大量指向驱动程序函数的指针。这段代码唯一不同寻常的地方是设置flags为IFF_NOARP。这表明接口不能使用地址解析协议（Address Resolution Protocol，ARP）。ARP是以太网的底层协议；它的作用是将IP地址转换为以太网的MAC（medium access control，介质访问控制）地址。由于*snull*模拟的“远程”系统根本是不存在的，因此没有必要应答从那里发来的ARP请求。*snull*使用另外的ARP实现会使代码复杂化，因此在这里将接口标记为不能处理ARP协议。将hard_header_cache设置为NULL也是出于类似的原因：在这个接口中禁止对ARP的缓存（由于根本不存在）。这个主题将在本章后面的“MAC地址解析”中详细讨论。

初始化代码还设置了一些用来处理传输超时的成员（tx_timeout和watchdog_timeo）。在本章后面的“传输超时”一节中，将完整介绍相关内容。

这里需要对net_device结构的一个成员——priv作进一步解释。该成员的作用和字符驱动程序中private_data指针的作用类似。但和fops->private_data不同，priv指针是与net_device结构一起分配的。出于性能和灵活性方面的考虑，不鼓励直接访问priv成员。当驱动程序需要访问私有数据指针时，应当使用*netdev_priv*函数。因此*snull*驱动程序中充满了类似以下代码的声明：

```
struct snull_priv *priv = netdev_priv(dev);
```

*snull*驱动程序为priv成员声明了snull_priv数据结构：

```
struct snull_priv {
    struct net_device_stats stats;
    int status;
    struct snull_packet *ppool;
    struct snull_packet *rx_queue; /* 输入数据包链表 */
    int rx_int_enabled;
    int tx_packetlen;
    u8 *tx_packetdata;
    struct sk_buff *skb;
    spinlock_t lock;
};
```

这个结构包含了一个net_device_stats结构的实例，它是保存接口统计信息的标准地方。下面的代码行分配和初始化dev->priv：

```
priv = netdev_priv(dev);
memset(priv, 0, sizeof(struct snull_priv));
spin_lock_init(&priv->lock);
snull_rx_ints(dev, 1); /* 允许接收中断 */
```

模块的卸载

在卸载*snull*模块时，没有什么特殊的事情需要完成。模块的清除函数只是注销接口，完成一些需要在清除函数中完成的事，然后释放net_device给系统：

```
void snull_cleanup(void)
{
    int i;

    for (i = 0; i < 2;  i++) {
        if (snull_devs[i]) {
            unregister_netdev(snull_devs[i]);
            snull_teardown_pool(snull_devs[i]);
            free_netdev(snull_devs[i]);
        }
    }
    return;
}
```

*unregister_netdev*函数从系统中删除了接口；*free_netdev*函数将*net_device*结构返回给了系统。如果还在什么地方有对该结构的引用，则它将继续存在，但是驱动程序并不需要关注这一点。一旦注销了接口，内核就不会再调用这个函数了。

值得注意的是，只有当设备被注销后，内部的清除函数（在*snull_teardown_pool*中执行）才能被执行。它必须在将net_device结构返回给系统之前执行；一旦调用了*free_netdev*，则不能再对设备或者私有数据区进行引用。

net_device结构细节

net_device结构位于网络驱动程序层的最核心地位，因此值得对它进行完整的描述。这个列表描述了其中的所有成员，但对读者来说这仅仅作为一个参考而不必记忆它们。在本章的剩余部分，只要在例子代码中使用到它们，就会对它们进行简要的描述，这样读者就不必要翻回去重新看了。

全局信息

net_device结构的第一部分包含以下的成员：

`char name[IFNAMSIZ];`

设备名称。如果被驱动程序设置的名称中包含%d格式化字符串，*register_netdev*将使用一个数字替换它，使之成为唯一的名字。分配的编号从零开始。

```
unsigned long state;
```

设备状态。这个成员中包含有若干标志。驱动程序通常无需直接操作这些标志，相反，内核提供了一组工具函数。在讲述驱动程序操作时，我们将讨论这些函数。

```
struct net_device *next;
```

指向全局链表下一个设备的指针。驱动程序不应该修改这个成员。

```
int (*init)(struct net_device *dev);
```

初始化函数。如果这个指针被设置了，则 *register_netdev* 将调用该函数完成对net_device结构的初始化。大多数现代的网络驱动程序不再使用这个函数了，相反，它们是在注册接口前完成初始化工作的。

硬件信息

下面的成员包含了相关设备的底层硬件信息。它们继承了早期Linux网络的特点；大多数现代的驱动程序仍然使用它们（和 if_port 一起使用）。在这里列出它们以保证完整性。

```
unsigned long rmem_end;
unsigned long rmem_start;
unsigned long mem_end;
unsigned long mem_start;
```

设备内存信息。这些成员保存了设备使用的共享内存之起始和终止地址。如果该设备具有不同的接收和传输内存，则mem成员用于传输内存，而rmem成员用于接收内存。rmem成员从来不会在驱动程序本身之外被引用。根据约定，end成员的设置要保证 end-start 等于可用的板卡内存量。

```
unsigned long base_addr;
```

网络接口的I/O 基地址。这个成员和前述成员类似，要在设备探测阶段赋值。*ifconfig* 命令可显示或修改当前值。base_addr 也可在系统引导期间，或在装载期间在命令行显式赋值。和前面的内存成员类似，内核不会使用该成员。

```
unsigned char irq;
```

被赋予的中断号。在列出接口时，*ifconfig* 命令将打印 dev->irq 的值。这个值通常在引导或装载阶段设置，其后可利用 *ifconfig* 修改。

```
unsigned char if_port;
```

指定在多端口设备上使用哪个端口。举例来说，如果设备同时支持同轴电缆（IF_PORT_10BASE2）和双绞线（IF_PORT_10BASET）以太网连接时，可使用该成员。完整的已知端口类型在 *<linux/netdevice.h>* 中定义。

`unsigned char dma;`

为设备分配的DMA通道。该成员只对某些外设总线有用，比如ISA。除了用于显示信息（*ifconfig* 命令）之外，不会在设备驱动程序之外使用这个成员。

接口信息

大部分接口相关的信息可由*ether_setup*函数正确设置（或者针对其他硬件类型的setup函数）。以太网卡可利用这个通用函数设置大部分成员，但`flags`和`dev_addr`成员是设备特有的，因此必须在初始化期间显式赋值。

某些非以太网接口也可以使用类似*ether_setup*这样的辅助函数。*drivers/net/net_init.c*导出了一些类似的函数，如下所示：

`void ltalk_setup(struct net_device *dev);`

设置LocalTalk设备的函数。

`void fc_setup(struct net_device *dev);`

初始化光纤通道设备。

`void fddi_setup(struct net_device *dev);`

配置光纤分布式数据接口（Fiber Distributed Data Interface，FDDI）网络的接口。

`void hippi_setup(struct net_device *dev);`

初始化高性能并行接口（High-Performance Parallel Interface，HIPPI）的高速互连驱动程序的成员。

`void tr_setup(struct net_device *dev);`

处理令牌环网络接口的设置函数。

大多数设备都属于这些类中的一种。如果设备是一个崭新的类，就需要手工设置下面的成员了：

`unsigned short hard_header_len;`

硬件头的长度，即数据包中位于IP头，或者其他协议信息之前的octet数目。对以太网接口，`hard_header_len`的值是14（`ETH_HLEN`）。

`unsigned mtu;`

最大传输单元（MTU）。网络层使用该成员驱动数据包的传输。以太网的MTU是1500个octet（`ETH_DATA_LEN`）。

`unsigned long tx_queue_len;`

可在设备的传输队列中排队的最大帧数目。*ether_setup*将该成员设置为100，但也可以修改它。例如，为避免浪费系统内存，*plip*使用10（比起实际的以太网接口，*plip*的吞吐率要低些）。

`unsigned short type;`

接口的硬件类型。ARP使用`type`成员判断接口所支持的硬件地址类型。以太网接口的正确值是`ARPHRD_ETHER`，这也是*ether_setup*所设置的值。可识别的类型在*<linux/if_arp.h>*中定义。

```
unsigned char addr_len;
unsigned char broadcast[MAX_ADDR_LEN];
unsigned char dev_addr[MAX_ADDR_LEN];
```

硬件（MAC）地址长度以及设备的硬件地址。以太网地址长度是6个octet（即接口板卡的硬件ID），广播地址由6个`0xff` octet组成。*ether_setup*会对上述值进行正确的设置。另一方面，设备地址必须从接口板卡中以设备特有的方式读取，因此驱动程序要负责将该地址复制到`dev_addr`。在数据包交给驱动程序传输之前，要利用硬件地址生成正确的以太网数据包头。*snull*不使用物理接口，从而使用的是它自己设定的硬件地址。

```
unsigned short flags;
int features;
```

接口标志（下面详述）。

该标志成员是一个包含如下位值的位掩码。`IFF_`前缀表示“接口标志“。某些标志由内核管理，而其他一些则由接口在初始化期间设置，用来声明接口的各种能力及其他特性。有效的标志定义在*<linux/if.h>*中，解释如下：

`IFF_UP`

对驱动程序，该标志只读。当接口被激活并可以开始传输数据包时，内核设置该标志。

`IFF_BROADCAST`

该标志（为网络代码所维护）说明接口允许广播。以太网卡是可广播的。

`IFF_DEBUG`

表示调试模式。该标志可用来控制用于调试目的的大量*printk*调用。尽管目前还没有正式的驱动程序使用该标志，但用户程序可通过ioctl设置或清除该标志，因此驱动程序可以利用这个标志。*mics-progs/netifdebug*程序可用来打开或关闭该标志。

IFF_LOOPBACK

该标志只能对回环设备进行设置。内核检查IFF_LOOPBACK标志以判断接口是否为回环设备，而不是将lo作为特殊的接口名称进行判断。

IFF_POINTOPOINT

该标志表明接口连接到点对点链路。这个标志由驱动程序设置，有时也由*ifconfig*设置。例如，*plip*和PPP驱动程序将设置该标志。

IFF_NOARP

该标志表明接口不能执行ARP。例如，点对点接口不需要运行ARP，因为如果运行了ARP，不但不能获得有用的信息，而且增加了网络传输量。*snull*缺少ARP功能，因此设置了这个标志。

IFF_PROMISC

设置该标志（由网络代码完成）将激活混杂模式。默认情况下，以太网接口使用一个硬件过滤器来确保它只接收广播数据包，以及直接发送到接口硬件地址的数据包。像*tcpdump*这样的数据包侦听器（sniffer）会在接口上设置混杂模式，以便检索到通过传输介质的所有数据包。

IFF_MULTICAST

该标志由驱动程序设置，表示该接口能够进行组播（multicast）发送。*ether_setup*默认设置IFF_MULTICAST，因此如果驱动程序不支持组播，就必须在初始化时清除该标志。

IFF_ALLMULTI

该标志告诉接口接收所有的组播数据包。仅仅在IFF_MULTICAST被设置的情况下，内核在主机执行组播路由时设置该标志。IFF_ALLMULT对接口来讲是只读的。我们将在本章后面的“组播”一节中看到组播标志的使用。

IFF_MASTER

IFF_SLAVE

该标志由负载均衡代码使用。接口驱动程序无需了解该标志。

IFF_PORTSEL

IFF_AUTOMEDIA

该标志表明设备能够在多种介质类型之间切换，例如，在非屏蔽双绞线（UTP）和同轴以太网电缆之间。如果IFF_AUTOMEDIA被设置，设备会自动选择正确的介质类型。在实际使用中，这两个标志都不会被内核使用。

IFF_DYNAMIC

该标志由驱动程序设置，表示接口地址可改变。现在内核不使用该标志。

IFF_RUNNING

该标志表示接口已经启动并且正在运行。该标志主要用于BSD兼容性，内核很少使用该标志。大多数网络驱动程序不需要关心IFF_RUNNING标志。

IFF_NOTRAILERS

Linux不使用该标志，只是为了和BSD兼容。

在程序改变IFF_UP时，会调用*open*或*stop*设备函数。当IFF_UP或其他任意一个标志被修改时，*set_multicast_list*函数将被调用。如果驱动程序需要在标志被修改时执行一些动作，则必须在*set_multicast_list*中完成这些动作。例如，当IFF_PROMISC被设置或清除，*set_multicast_list*必须通知板卡上的硬件过滤器。“组播”一节中将讲述该设备方法的职责。

驱动程序设置net_device结构中的功能成员，以告诉内核该接口硬件的特殊功能。本章将讨论这些功能的一部分，而其余功能超出了本书讨论的范围。完整的功能列出如下：

NETIF_F_SG

NETIF_F_FRAGLIST

这两个标志控制了分散/聚集I/O的使用。如果一个数据包被分成了多个独立的内存段，而接口又能传输这样的数据包，则需要设置NETIF_F_SG。当然必须实现分散/聚集I/O（将在“分散/聚集I/O”一节中讨论）。NETIF_F_FRAGLIST表明接口能够处理那些被分成块的数据包，在内核2.6中只有回环设备有此功能。

请注意如果内核不能提供检验的话，也不能为驱动程序执行分散/聚集I/O。原因是：如果内核忽略了一个数据包片段（“非线性”）而去计算校验值，它也能同时拷贝数据并将它们结合起来。

NETIF_F_IP_CSUM

NETIF_F_NO_CSUM

NETIF_F_HW_CSUM

这些标志告诉内核，不要对通过接口传出系统的部分或者全部的数据包使用校验。如果接口仅仅能够校验IP数据包，则设置NETIF_F_IP_CSUM。如果该接口不需要校验，则设置NETIF_F_NO_CSUM。回环设备设置该标志，*snull*也设置了它，这是因为数据包只是通过系统内存传输，因此不会失败，也就不需要检查它们了。如果硬件自己进行校验的话，则设置NETIF_F_HW_CSUM。

NETIF_F_HIGHDMA

如果设备可以在高端内存使用DMA，设置该标志。如果不设置该标志，所有为驱动程序提供的数据包缓冲区将在低端内存中分配。

NETIF_F_HW_VLAN_TX

NETIF_F_HW_VLAN_RX

NETIF_F_HW_VLAN_FILTER

NETIF_F_VLAN_CHALLENGED

这些选项表示硬件支持802.1q VLAN数据包。对VLAN的支持已经超出了本章的范围。如果设备不支持VLAN，设置NETIF_F_VLAN_CHALLENGED标志。

NETIF_F_TSO

如果设备能够执行TCP分割卸载，则设置该标志。TSO是一个新的特性，本章将不予讨论。

设备方法

和字符及块设备类似，每个网络设备都要声明作用其上的函数。本节将给出可在网络接口上执行的操作，某些操作可保留为NULL，其他一些无需修改，因为*ether_setup*将赋予适当的方法。

网络接口的设备方法可划分为两个类型：基本的和可选的。基本方法包括使用接口必需的方法；可选方法实现了一些更为高级的功能，但并不严格要求有这些方法。下面是基本方法：

```
int (*open)(struct net_device *dev);
```

打开接口。在*ifconfig*激活接口时，接口将被打开。*open*函数应该注册所有的系统资源（I/O端口、IRQ、DMA等等），打开硬件，并对设备执行其他所需的设置。

```
int (*stop)(struct net_device *dev);
```

停止接口。当接口终止时应该被停止。在该函数中执行的操作与打开时执行的操作相反。

```
int (*hard_start_xmit) (struct sk_buff *skb, struct net_device *dev);
```

该方法初始化数据包的传输。完整的数据包（协议头和数据）包含在一个套接字缓冲区（sk_buffer）结构中。套接字缓冲区将在本章后面介绍。

```
int (*hard_header) (struct sk_buff *skb, struct net_device *dev, unsigned
  short type, void *daddr, void *saddr, unsigned len);
```

该函数（在*hard_start_xmit*前被调用）根据先前检索到的源和目标硬件地址建立硬件头。该函数的任务是将作为参数传递进入的信息，组织成设备特有的适当硬件头。*eth_header*是以太网类型接口的默认函数，*ether_setup*将该成员赋值成*eth_header*。

```
int (*rebuild_header)(struct sk_buff *skb);
```
该函数用来在传输数据包之前、完成ARP解析之后，重新建立硬件头。以太网设备使用的默认函数使用ARP填充数据包中缺少的信息。

```
void (*tx_timeout)(struct net_device *dev);
```
如果数据包的传输在合理的时间段内失败，则假定丢失了中断或接口被锁住，这时网络代码将调用该方法。它负责解决问题并重新开始数据包的传输。

```
struct net_device_stats *(*get_stats)(struct net_device *dev);
```
当应用程序需要获得接口的统计信息时，将调用该函数。例如，在运行*ifconfig*或*netstat -i*命令时将利用该方法。我们将在本章后面的“统计信息中”看到*snull*的样例实现。

```
int (*set_config)(struct net_device *dev, struct ifmap *map);
```
改变接口配置。该函数是配置驱动程序的入口点。利用*set_config*，可在运行中改变设备的I/O地址和中断号。在探测不到接口时，系统管理员可使用该函数。现代硬件的驱动程序通常不需要实现该方法。

其余的设备操作可看作是可选的:

```
int weight;
int (*poll)(struct net_device *dev; int *quota);
```
NAPI兼容驱动程序提供该方法，在禁止中断时，以轮询模式操作接口。NAPI（以及weight成员）将在“不使用接收中断”一节中讲述。

```
void (*poll_controller)(struct net_device *dev);
```
该函数在禁止中断的情况下，要求驱动程序在接口上检查事件。它被用于特定的内核网络任务中，比如远程控制台和内核网络调试。

```
int (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);
```
执行接口特有的*ioctl*命令（本章后面的“定制ioctl命令”中描述了这些命令的实现）。如果接口不需要实现任何接口特有的命令，则net_device中对应的成员可保持为NULL。

```
void (*set_multicast_list)(struct net_device *dev);
```
当设备的组播列表发生改变，或者设备标志发生改变时，将调用该方法。“组播”一节将详细描述该方法，并给出一个样例实现。

```
int (*set_mac_address)(struct net_device *dev, void *addr);
```
如果接口支持硬件地址的改变，则可实现该方法。许多接口根本不支持这种功能。其他接口使用默认的*eth_mac_addr*实现（在*drivers/net/net_init.c*中定义）。

*eth_mac_addr*仅仅将新地址复制到dev->dev_addr中，而且只能在接口不工作时进行设置。使用*eth_mac_addr*的驱动程序应该在open函数中，用dev->dev_addr设置硬件的MAC地址。

```
int (*change_mtu)(struct net_device *dev, int new_mtu);
```

在接口的MTU（maximum transfer unit，最大传输单元）改变时，该函数负责采取相应的动作。如果驱动程序在用户改变MTU时需要完成某些特定工作，则应该声明自己的函数，否则默认的函数可正确实现相关处理。*snull*实现了该方法，可作为模板参考。

```
int (*header_cache) (struct neighbour *neigh, struct hh_cache *hh);
```

*head_cache*将根据ARP查询的结果填充hh_cache结构。几乎所有的驱动程序都可以使用默认的*eth_header_cache*实现。

```
int (*header_cache_update) (struct hh_cache *hh, struct net_device *dev,
  unsigned char *haddr);
```

在发生变化时，该方法更新hh_cache结构中的目标地址。以太网设备使用*eth_header_cache_update*。

```
int (*hard_header_parse) (struct sk_buff *skb, unsigned char *haddr);
```

*hard_header_parse*函数从skb中包含的数据包中获得源地址，并将其复制到位于haddr的缓冲区。该函数的返回值是地址的长度。以太网设备通常使用*eth_header_parse*。

工具成员

net_device结构中其余的数据成员由接口使用，保存一些有用的状态信息。某些成员由*ifconfig*和*netstat*使用，以便为用户提供当前的配置信息。因此接口应该给这些成员赋予适当的值：

```
unsigned long trans_start;
unsigned long last_rx;
```

这些成员都保存一个jiffies值。驱动程序分别在传输开始及接收到数据包时负责更新这些值。网络子系统使用trans_stat值检测传输器是否被锁住。last_rx当前未使用，但驱动程序应该维护这个成员，以便将来使用。

```
int watchdog_timeo;
```

在网络层确定传输已经超时，并且调用驱动程序的*tx_timeout*函数之前的最小时间（jiffies为单位）。

`void *priv;`

和`filp->private_data`等价。在现代的驱动程序中，该成员由*alloc_netdev*设置，并且不能被直接访问；如要访问，需要使用*netdev_priv*函数。

`struct dev_mc_list *mc_list;`

`int mc_count;`

上面这两个成员用来处理组播传输。`mc_count`是`mc_list`所包含的项的数目。详细信息，可参阅“组播”一节。

`spinlock_t xmit_lock;`

`int xmit_lock_owner;`

`xmit_lock`用来避免同时对驱动程序的*hard_start_xmit*函数的多次调用。`xmit_lock_owner`是获得`xmit_lock`的CPU编号。驱动程序不应改变这些成员。

`net_device`结构中还有其他一些成员，但网络驱动程序不使用它们。

打开和关闭

驱动程序可在装载阶段或内核引导阶段探测接口。但是在接口能够传送数据包之前，内核必须打开接口并赋予其地址。内核可在响应*ifconfig*命令时打开或关闭一个接口。

在使用*ifconfig*向接口赋予地址时，要执行两个任务。首先，它通过`ioctl(SIOCSIFADDR)`(Socket I/O Control Set Interface Address)赋予地址，然后通过`ioctl(SIOCSIFFLAGS)`(Socket I/O Control Set Interface Flags)设置`dev->flag`中的`IFF_UP`标志以打开接口。

对设备而言，无需对`ioctl(SIOCSIFADDR)`做任何工作。内核不会调用任何驱动程序函数，也就是说，该任务由内核来执行，是与设备无关的。而后一个命令(`ioctl(SIOCSIFFLAGS)`)会调用设备的*open*方法。

类似地，在接口被关闭时，*ifconfig*使用`ioctl(SIOCSIFFLAGS)`来清除`IFF_UP`标志，然后调用*stop*函数。

这两个设备函数在成功时均返回0，而在失败时和通常一样，返回负值。

对实际代码而言，驱动程序必须执行许多和字符及块设备相同的任务。*open*请求必要的系统资源，并告诉接口开始工作；*stop*关闭接口并释放系统资源。但是除此之外，还要执行其他一些步骤。

首先，在接口能够和外界通讯之前，要将硬件地址（MAC）从硬件设备复制到dev->dev_addr。硬件地址可在打开期间拷贝到设备中。*snull* 软件接口在 *open* 时赋予硬件地址 —— 它其实使用了一个长度为ETH_ALEN的ASCII字符串作为假的硬件地址，其中 ETH_ALEN 是以太网硬件地址的长度。

一旦准备好开始发送数据后，*open* 方法还应该启动接口的传输队列（允许接口接受传输数据包）。内核提供的如下函数可启动该队列：

```
void netif_start_queue(struct net_device *dev);
```

snull 的 *open* 代码如下所示：

```
int snull_open(struct net_device *dev)
{
    /* request_region(), request_irq(), ....  (like fops->open) */

    /*
     * 对主板的硬件地址赋值：使用 "\0SNULx"，
     * 其中x是0或者1。第一个字节是 '\0' 是
     * 为了避免成为组播地址（组播的addrs第一个字节是奇数）。
     */
    memcpy(dev->dev_addr, "\0SNUL0", ETH_ALEN);
    if (dev = = snull_devs[1])
        dev->dev_addr[ETH_ALEN-1]++; /* \0SNUL1 */
    netif_start_queue(dev);
    return 0;
}
```

正如读者已经看到的那样，在缺少实际硬件的情况下，在*open*函数要做的事情很少。对 *stop* 函数而言，也是这样，它只是 *open* 的反操作。出于这个原因，实现 *stop* 的函数经常被称为 *close* 或 *release*。

```
int snull_release(struct net_device *dev)
{
    /* 释放端口、irq及如fops->close中释放的东西 */

    netif_stop_queue(dev); /* 不能再继续传输 */
    return 0;
}
```

函数：

```
void netif_stop_queue(struct net_device *dev);
```

是 *netif_start_queue* 的逆操作，它标记设备不能传输其他数据包。在接口被关闭时（在 *stop* 函数中），必须调用该函数，但该函数也可以用来临时停止传输，将在下一节讲述相关内容。

数据包传输

网络接口所执行的最重要任务是数据的传输和接收。这里首先讨论传输，因为数据的传输相对容易理解一些。

传输指的是将数据包通过网络连接发送出去的行为。无论何时内核要传输一个数据包，它都会调用驱动程序的*hard_start_transmit*函数将数据放入外发队列。内核处理的每个数据包位于一个套接字缓冲区结构（sk_buff）中，该结构定义在 *<linux/skbuff.h>* 中。这个结构的名称来自于表示网络连接的Unix抽象，即套接字（socket）。尽管接口无需处理套接字，但每个网络数据包属于更高网络层的某个套接字，而且所有套接字的输入/输出缓冲区都是sk_buff结构形成的链表。同一个sk_buff结构还用在主机网络数据以及所有的Linux网络子系统，但是对接口而言，套接字缓冲区仅仅是一个数据包而已。

指向 sk_buff 的指针通常称为 skb，因此在代码和正文中将继续这个叫法。

套接字缓冲区是一个复杂的结构，内核提供了许多用来操作该结构的函数。我们将在“套接字缓冲区”一节中描述这些函数，现在只需了解一些关于sk_buff的基本概念，就能编写一个可工作的驱动程序。

传递给*hard_start_xmit*的套接字缓冲区包含了物理数据包（以它在介质上的格式），并拥有完整的传输层数据包头。接口无需修改要传输的数据。skb->data指向要传输的数据包，而skb->len是以octet为单位的长度。如果驱动程序能够处理scatter/gather I/O，形势将变得有些复杂；本章将在“Scatter/Gather I/O”一节中详细讲述。

下面是*snull*的数据包传输代码。实现传输的物理机制在另外一个单独的函数中实现，这是因为每个接口驱动程序都必须根据其驱动的特有硬件实现这段代码：

```
int snull_tx(struct sk_buff *skb, struct net_device *dev)
{
    int len;
    char *data, shortpkt[ETH_ZLEN];
    struct snull_priv *priv = netdev_priv(dev);

    data = skb->data;
    len = skb->len;
    if (len < ETH_ZLEN) {
        memset(shortpkt, 0, ETH_ZLEN);
        memcpy(shortpkt, skb->data, skb->len);
        len = ETH_ZLEN;
        data = shortpkt;
    }
    dev->trans_start = jiffies; /* 保存时间戳 */

    /* 记住skb，这样可以在中断时刻释放它 */
```

```
    priv->skb = skb;

    /* 对数据的实际传送是设备相关的，并不在这里显示 */
    snull_hw_tx(data, len, dev);

    return 0; /* 该设备不可能失败 */
}
```

该传输函数只执行了对数据包的一致性检查，然后通过硬件相关的函数传输数据。当所需传输的数据包长度小于介质（对*snull*来说就是虚拟网卡）所支持的最小长度时，则需小心处理。许多Linux网络驱动程序（对其他操作系统的网络驱动程序也一样）在遇到这种情况时，会漏掉一些数据。为了克服这个安全漏洞，需要将短小的数据包拷贝到一个分立的数组中，而对这个数组，则又可以对介质所支持的全部长度个元素清零（由于最小长度是60个字节，实在太小了，所以可以将数据做压栈处理）。

如果执行成功，则*hard_start_xmit*返回0。此时负责传输数据包的驱动程序要尽量保证传输的正常进行，而且完毕后必须释放skb。返回一个非零值表示此次数据传输失败；内核将会重试传输。在这种情况下，驱动程序应该在解决掉导致失败的原因前，停止队列。

这里忽略了“硬件相关”的传输函数（*snull_hw_tx*），是因为其内部尽是*snull*设备的欺骗性代码（包括操作源和目标地址），从而对实际网络驱动程序编写者来讲意义不大。当然，如果读者对此感兴趣，也可以从示例代码中看到这个函数的完整实现。

控制并发传输

*hard_start_xmit*函数通过net_device结构中的一个自旋锁（*xmit_lock*）获得并发调用时的保护。但是在该函数返回后，有可能再次被调用。当软件指示硬件开始传输数据包之后，该函数返回，但是硬件传输可能尚未结束。这对*snull*来说不是问题，因为它利用CPU完成所有的工作，因此在传输函数返回时，数据包的传输已经结束。

另一方面，实际的硬件接口却是异步传输数据包的，而且可用来保存外发数据包的存储空间非常有限。在内存被耗尽时（对某些硬件，也许单个外发数据包的传输就会使内存耗尽），驱动程序需要告诉网络系统在硬件能够接受新数据之前，不能启动其他的数据包传输。

调用*netif_stop_queue*可完成这一通知。前面在停止队列时介绍过这个函数。在驱动程序停止队列之后，它必须在将来的某个时刻，当它能够再次接受数据包的传输时，重新启动该队列。为此，应调用：

```
void netif_wake_queue(struct net_device *dev);
```

这个函数除了通知网络系统可再次开始传输数据包以外，和*netif_start_queue*函数一样。

许多现代的网络接口在传输多个数据包时，维护一个内部的队列，这样可以获得最好的网络性能。这种设备的网络驱动程序在任意时刻都可支持多个外发传输数据包，但不管设备是否支持多个外发传输数据包，设备内存都会被填满。一旦设备内存填充到容不下最大可能的数据包的时候，驱动程序应该停止队列，直到空间再次可用为止。

如果想从其他地方，而不是从*hard_start_xmit*函数（可能负责重新配置请求）中禁止数据包的传送，则要调用下面的函数：

```
void netif_tx_disable(struct net_device *dev);
```

该函数的行为与*netif_stop_queue*类似，但它还确保了在返回时，在其他的CPU上没有运行*hard_start_xmit*函数。通常可以使用*netif_wake_queue*函数再次开始传输队列。

传输超时

大部分处理实际硬件的驱动程序必须能够应付硬件偶尔不能正确响应的问题。接口也许会忘记它在做什么，或者系统有可能丢失中断。这种类型的问题在个人计算机上的某些设备中很常见。

许多驱动程序利用定时器处理这类问题；如果某个操作在定时器到期时还未完成，则认为出现了问题。从本质上讲，网络系统是通过大量定时器控制的多个状态机的复杂组合。从这个角度上讲，网络代码把检测传输超时作为其常用操作之一。

因此网络驱动程序无需自己检测这种问题。相反驱动程序只需设置一个超时周期，并在net_device结构的watchdog_timeo成员中设置。这个周期以jiffies为单位，对通常的传输延迟（比如网络介质上因堵塞造成的冲突）来讲应该是足够长的。

如果当前的系统时间超过设备的trans_start时间至少一个超时周期，网络层将最终调用驱动程序的*tx_timeout*函数。这个函数的任务是完成解决超时问题而需要的任何工作，并确保正在进行的任何传输能够正常结束。驱动程序不能丢失网络代码提交的套接字缓冲区，这一点尤其重要。

*snull*可模拟这种传输器被锁住的情形，可通过装载时的两个参数控制：

```
static int lockup = 0;
module_param(lockup, int, 0);

static int timeout = SNULL_TIMEOUT;
module_param(timeout, int, 0);
```

如果使用lockup=n参数装载驱动程序，则会在传输n个数据包之后模拟一次传输器硬件的锁住，并且watchdog_timeo成员将设置为给定的超时值。在模拟锁住时，*snull*会调用*netif_stop_queue*以避免产生其他的传输请求。

*snull*的传输超时处理器程序如下所示：

```
void snull_tx_timeout (struct net_device *dev)
{
    struct snull_priv *priv = netdev_priv(dev);
    PDEBUG("Transmit timeout at %ld, latency %ld\n", jiffies,
            jiffies - dev->trans_start);
        /* 模拟一个传输中断 */
    priv->status = SNULL_TX_INTR;
    snull_interrupt(0, dev, NULL);
    priv->stats.tx_errors++;
    netif_wake_queue(dev);
    return;
}
```

当传输超时发生时，驱动程序必须在接口统计信息中标记该错误，并要将设备重置为一个合理的状态，以便传输新的数据包。在*snull*中发生超时时，驱动程序调用*snull_interrupt*填补“丢失”的中断，并调用*netif_wake_queue*重新启动传输队列。

Scatter/Gather I/O

在网络上为传输工作创建数据包的过程，包括了组装多个数据片段的过程。数据包中的数据需要从用户空间拷贝出来，并且针对各种不同层次的网络协议栈添加数据头。这个组装过程包含了大量的数据拷贝工作。如果负责发送数据包的网络接口实现了分散/聚集I/O，则数据包就不用组装成一个大的数据块，而且省去了许多拷贝操作。分散/聚集I/O还能用“零拷贝”的方法，把网络数据直接从用户空间缓冲区内传输出来。

只有在device结构中的feature成员内设置了NETIF_F_SG标志位，内核才将分散的数据包传递给*hard_start_xmit*函数。如果设置了该标志，还需要检查skb中的“共享信息”成员，以判断数据包是由一个数据片段组成，还是由大量数据片段组成，并且负责查找所需要的分散数据片段。这使用一个特殊的、称之为*skb_shinfo*的宏来访问这个信息。使用下面的代码作为传输潜在数据包片段的第一步：

```
if (skb_shinfo(skb)->nr_frags =  = 0) {
    /* 像通常一样只使用skb->data和skb->len */
}
```

nr_frags成员表明组成数据包的数据片段个数。如果它是0，则说明数据包由一个数据片段组成，可以通过data成员来访问。如果它不是0，则驱动程序必须检查和组装

每个要传输的数据片段。skb结构中的data成员指向了第一个数据片段（相对于那些由一个数据片段组成的数据包）。数据片段的长度必须通过在skb->len（它依然保存了全部数据包的长度）中减去skb->data_len来得到。剩余的数据片段保存在共享数据结构的frags的数组中。frags的每个入口都是一个skb_frag_struct结构：

```
struct skb_frag_struct {
    struct page *page;
    __u16 page_offset;
    __u16 size;
};
```

由此可见，这次又要处理页结构，而不是内核虚拟地址了。驱动程序将遍历数据片段，并为DMA传输进行映射，这些工作也发生在直接放在skb结构中的第一个数据片段上。当然硬件负责组装数据片段，并把它们作为一个单独的数据包发送出去。请注意，如果设置了NETIF_F_HIGHDMA标志，其中一些，或者是所有的数据片段可能放在了高端内存区。

数据包的接收

从网络上接收数据要比传输数据复杂一点，这是因为必须在原子上下文中分配一个sk_buff并传递给上层处理。网络驱动程序实现了两种模式接收数据包：中断驱动方式和轮询方式。大多数驱动程序实现了中断驱动技术，也是本章讨论的第一种模式。一些宽带适配器的驱动程序也实现了轮询技术，将在“不使用接收中断”一节中讲到。

*snull*的实现从设备无关的管理工作中将“硬件”细节隔离了出来。*snull_rx*函数在硬件接收到数据包之后（数据包已经在计算机内存中了），被*snull*的“中断”处理程序调用。*snull_rx*接收一个指向数据的指针，以及数据包的长度。它还负责将数据包以及其他附加信息发送到上层的网络代码。这一代码和获得数据指针及其长度的途径无关。

```
void snull_rx(struct net_device *dev, struct snull_packet *pkt)
{
    struct sk_buff *skb;
    struct snull_priv *priv = netdev_priv(dev);

    /*
     * 已经从传输介质中获得了数据包。建立封装它的skb，使得上层可以处理它
     */
    skb = dev_alloc_skb(pkt->datalen + 2);
    if (!skb) {
        if (printk_ratelimit())
            printk(KERN_NOTICE "snull rx: low on mem - packet dropped\n");
        priv->stats.rx_dropped++;
        goto out;
    }
```

```
        memcpy(skb_put(skb, pkt->datalen), pkt->data, pkt->datalen);

        /* 写入数据、然后将其传递给接收层 */
        skb->dev = dev;
        skb->protocol = eth_type_trans(skb, dev);
        skb->ip_summed = CHECKSUM_UNNECESSARY; /* 不必检查它 */
        priv->stats.rx_packets++;
        priv->stats.rx_bytes += pkt->datalen;
        netif_rx(skb);
      out:
        return;
    }
```

该函数十分通用，从而可以作为真实网络驱动程序的一个模板，但在使用这些代码段之前，需要一些必要的解释。

第一步是分配一个保存数据包的缓冲区。注意缓冲区的分配函数（*dev_alloc_skb*）需要知道数据长度。这个函数利用这一信息为缓冲区分配空间。*dev_alloc_skb*以原子的优先权调用*kmalloc*，因此可在中断期间安全使用。内核提供了分配套接字缓冲区的其他接口，但不值得在这里讨论。本章后面的“套接字缓冲区”一节中将详细解释套接字缓冲区。

当然必须检查*dev_alloc_skb*函数的返回值，*snull*也这样做了。可以在失败时使用*printk_ratelimit*。每秒钟产生成百上千条控制台消息是使系统整体性能下降的好方法，并且会隐藏产生问题的真实原因；当向控制台发送大量信息时，*printk_ratelimit*返回0，为预防该情况的发生，当然性能会有所下降。

一旦拥有一个合法的skb指针，则调用*memcpy*将数据包数据拷贝到缓冲区内。*skb_put*函数刷新缓冲区内的数据末尾指针，并且返回新创建数据区的指针。

如果正在为某个接口编写能够实现完整总线控制I/O的高性能驱动程序，则可以考虑一种优化可能性。某些驱动程序在传入数据包到达之前为它们分配套接字缓冲区，然后指示接口直接将数据包数据放入套接字缓冲区中。网络层与这种策略相配合，会在可进行DMA的空间中分配所有的套接字缓冲区（如果设备设置了NETIF_F_HIGHDMA标志，它可能在高端内存中）。这样可避免填充套接字缓冲区的额外复制操作，但是因为无法预先知道传入的数据包大小，所以必须谨慎处理缓冲区大小。在这种情况下，*chang_mtu*方法的实现也很重要，因为它可以让驱动程序对最大数据包大小的改变做出适当的响应。

在能够处理数据包之前，网络层必须知道数据包的一些信息。为此必须在将缓冲区传递到上层之前，对dev和protocol成员正确赋值。以太网支持代码导出了辅助函数*eth_type_trans*，用来查找填入protocol中的正确值。然后需要指定如何求得校验和，或者已经在数据包上求得了校验和（*snull*无需求得任何校验和）。skb->ip_summed的可能策略如下所示：

CHECKSUM_HW

设备已经在硬件层求得了校验和。SPARC HME 接口是硬件校验和的一个例子。

CHECKSUM_NONE

校验和还未被验证，而且该任务必须由系统软件完成。对新分配的缓冲区，这是默认策略。

CHECKSUM_UNNECESSARY

不进行任何校验和的计算。这是 *snull* 和回环接口的策略。

读者可能会问：已经在net_device结构的features成员中设置了标志，为什么这里还必须指定校验和呢？原因是features标志告诉内核设备是如何处理传出的数据包。对传入的数据包并不使用该标志，因此必须单独设置它。

最后，驱动程序更新其统计计数器，以记录已接收到每个数据包。统计结构中包含若干成员，最重要的有rx_packets、rx_bytes、tx_packets和tx_bytes，其中包含了已接收和已发送的数据包个数，以及已传输的octet总量。本章后面的“统计信息”一节中将全面介绍所有的成员。

接收数据包过程中的最后一个步骤由 *netif_rx* 执行，它将套接字缓冲区传递给上层软件处理。实际上 *netif_rx* 返回一个整型值；NET_RX_SUCCESS (0)表示数据包已经被成功接收；其他的值表示失败。有三个返回值（NET_RX_CN_LOW、NET_RX_CN_MOD 和 NET_RX_CN_HIGH)由低到高表示网络子系统的拥堵状况；NET_RX_DROP表示对数据包的丢失。当拥堵严重时，驱动程序使用这些值终止向内核发送数据包，但是在实际的操作中，大多数驱动程序忽略从 *netif_rx* 函数返回的值。如果读者正在编写一个宽带设备的驱动程序，并且希望对网络堵塞情况做出正确的响应，最好的方法是实现NAPI，在讲述中断处理例程之后将讲到它。

中断处理例程

大多数硬件接口通过中断处理例程来控制。接口在两种可能的事件下中断处理器：新数据包到达，或者外发数据包的传输已经完成。网络接口还能产生中断以通知错误的产生、连接状态等等情况。

通常中断例程通过检查物理设备中的状态寄存器，以区分新数据包到达中断和数据传输完毕中断。*snull* 接口工作原理与此类似，但是它的状态值是通过软件方法实现的，其保存在dev->priv中。网络设备的中断处理例程类似于如下代码：

```
static void snull_regular_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
```

```
    int statusword;
    struct snull_priv *priv;
    struct snull_packet *pkt = NULL;
    /*
     * 同往常一样，检查“device”指针以确保它指向了中断。
     * 然后为“struct device *dev”赋值
     */
    struct net_device *dev = (struct net_device *)dev_id;
    /* ... 检查硬件以确保中断的确是发给我们的 */

    /* 超过正常范围 */
    if (!dev)
        return;

    /* 锁住设备 */
    priv = netdev_priv(dev);
    spin_lock(&priv->lock);

    /* 获得状态字：真实的网络设备使用I/O指令 */
    statusword = priv->status;
    priv->status = 0;
    if (statusword & SNULL_RX_INTR) {
        /* 将其发送给snull_rx进行处理 */
        pkt = priv->rx_queue;
        if (pkt) {
            priv->rx_queue = pkt->next;
            snull_rx(dev, pkt);
        }
    }
    if (statusword & SNULL_TX_INTR) {
        /* 一个传输过程完毕：释放skb */
        priv->stats.tx_packets++;
        priv->stats.tx_bytes += priv->tx_packetlen;
        dev_kfree_skb(priv->skb);
    }

    /* 为设备解锁，并结束处理 */
    spin_unlock(&priv->lock);
    if (pkt) snull_release_buffer(pkt); /* Do this outside the lock! */
    return;
}
```

该处理例程的第一个任务是检索指向正确 struct net_device 的指针。该指针通常来自以参数形式接收到的 dev_id 指针。

该处理程序有意思的部分是对“传输结束”情形的处理。在这种情况下，统计信息要被更新，而且要调用 *dev_kfree_skb* 将（不再使用的）套接字缓冲区返回给系统。实际上有三个类似的函数调用：

`dev_kfree_skb(struct sk_buff *skb);`

如果知道代码不在中断上下文中运行则调用该函数。由于*snull*没有实际的硬件中断，所以使用该函数。

`dev_kfree_skb_irq(struct sk_buff *skb);`

如果要在中断处理例程中释放缓冲区，则使用该函数。此时使用该函数能优化系统性能。

`dev_kfree_skb_any(struct sk_buff *skb);`

如果相关代码既可以在中断上下文中运行，也能在非中断上下文中运行，使用该函数。

最后，如果驱动程序暂时终止了传输队列，同时使用*netif_wake_queue*重新启动传输队列。

与传输相反，数据包的接收不需要其他任何特殊的中断处理。调用*snull_rx*（已经介绍过这个函数）就足够了。

不使用接收中断

用上面描述的方法编写一个网络驱动程序的话，当接口收到每一个数据包时，处理器都会被中断。在许多情况下，这是希望的操作模式，不会引起任何问题。然而宽带接口每秒钟会收到几千个数据包。使用这种中断方式，会使系统性能全面下降。

为了能提高Linux在宽带系统上的性能，网络子系统开发者创建了另外一种基于轮询方式的接口（称之为NAPI，注1）。在驱动程序开发者中，轮询的名声并不好，他们经常视轮询为一种低效和无能的技术。然而轮询的低效率仅表现在接口没事情可做时。当系统中存在一个处理大流量的高速接口时，每个时刻都有大量的数据包需要处理。此时没有必要中断处理器；使用轮询技术处理到达接口的每一个数据包就足够了。

停止使用中断会减轻处理器的负荷。由于网络堵塞，数据包处于网络代码中时，NAPI型的驱动程序将不会把数据包发送给内核，这将能提高系统的性能。出于多种原因，NAPI驱动程序也很少重新排序数据包。

并不是所有的设备都能在NAPI模式下工作。一个NAPI接口必须能够保存多个数据包（或者在板卡上，或者在内存的DMA环中）。接口对接收数据包能够禁止中断，同时又

注1： NAPI表示“新API”，比起取名，网络黑客更擅长创建接口。

能对传输和其他事件打开中断。还有一些其他的问题导致编写一个NAPI驱动程序非常困难；参看内核代码树中的*Documentation/networking/NAPI_HOWTO.txt*，可了解详细情况。

只有很少的驱动程序实现了NAPI接口。如果正在为一个能产生大量中断的接口编写驱动程序，花点时间实现NAPI是值得的。

当把use_napi参数设置为非零时装载*snull*驱动程序，则使用NAPI模式操作。在初始化时，必须对一对额外的structnet_device成员进行设置：

```
if (use_napi) {
    dev->poll        = snull_poll;
    dev->weight      = 2;
}
```

poll成员必须设置为驱动程序的轮询函数，一会将讨论*snull_poll*函数。weight成员描述了接口的相对重要性：当资源紧张时，在接口上能承受多大的流量。对weight参数的设置没有严格的要求；一般来说，10MBps的以太网接口设置weight为16，而更快的接口设置为64。weight的值不能比接口所能保存的数据包数目大。在*snull*中，将weight设置为2，以说明延迟的数据包接收。

创建NAPI驱动程序中的下一步是修改中断处理例程。当接口（以激活接收中断方式启动）通知数据包到达的时候，中断程序不能处理该数据包。相反它还要禁止接收中断，并且告诉内核，从现在开始启动轮询接口。在*snull*的“中断”处理程序中，数据包接收中断代码被修改成如下形式：

```
if (statusword & SNULL_RX_INTR) {
    snull_rx_ints(dev, 0); /* 禁止今后的中断 */
    netif_rx_schedule(dev);
}
```

当接口通知数据包到来的时候，中断处理例程与接口相分离；此时需要做的所有事就是调用*netif_rx_schedule*函数，它负责在此后某个时间点调用*poll*函数。

轮询函数具有以下原型：

```
int (*poll)(struct net_device *dev, int *budget);
```

*snull*用以下代码实现了轮询函数：

```
static int snull_poll(struct net_device *dev, int *budget)
{
    int npackets = 0, quota = min(dev->quota, *budget);
    struct sk_buff *skb;
    struct snull_priv *priv = netdev_priv(dev);
```

```
    struct snull_packet *pkt;

    while (npackets < quota && priv->rx_queue) {
        pkt = snull_dequeue_buf(dev);
        skb = dev_alloc_skb(pkt->datalen + 2);
        if (! skb) {
            if (printk_ratelimit())
                printk(KERN_NOTICE "snull: packet dropped\n");
            priv->stats.rx_dropped++;
            snull_release_buffer(pkt);
            continue;
        }
        memcpy(skb_put(skb, pkt->datalen), pkt->data, pkt->datalen);
        skb->dev = dev;
        skb->protocol = eth_type_trans(skb, dev);
        skb->ip_summed = CHECKSUM_UNNECESSARY; /* 不必检查它 */
        netif_receive_skb(skb);

        /* 维护统计信息 */
        npackets++;
        priv->stats.rx_packets++;
        priv->stats.rx_bytes += pkt->datalen;
        snull_release_buffer(pkt);
    }
    /* 如果处理完所有的数据包，工作完成；告诉内核并且重新打开中断 */
     *budget -= npackets;
     dev->quota -= npackets;
     if (! priv->rx_queue) {
        netif_rx_complete(dev);
        snull_rx_ints(dev, 1);
        return 0;
    }
    /* 这里不能处理任何事 */
    return 1;
}
```

该函数的核心部分是创建包含数据包的skb；这段代码与以前的*snull_rx*相同。然而还是存在着一些区别：

- budget参数提供了能传递给内核的最大数据包数。在device结构中，quota成员给出了另外一个最大值；轮询函数必须接受二者之间的最小值。它还用实际接收的数据包数减小了dev->quota和*budget的值。bugdet值是当前CPU能够从所有接口中接收数据包的最大数目，而quota是在初始化阶段分配给接口的weight值。
- 用*netif_receive_skb*函数将数据包传递给内核，而不是使用*netif_rx*。
- 如果轮询函数能够处理在限制数量范围内的所有数据包，那么它将重新打开接收中断，调用*netif_rx_complete*关闭轮询函数，并返回0。如果返回值是1，表示数据包仍在被处理状态。

网络子系统确保在多于一个处理器的系统上，指定的轮询函数不会被同时调用。但是对于其他设备来说，对轮询函数的调用还可能是并发的。

链路状态的改变

网络连接要和本地系统之外的外界打交道。因此经常会受到外部事件的影响，而这些事件又可能是瞬时发生的。网络子系统需要了解网络链路是否正常，因而提供了驱动程序可以利用的几个函数。

大多数涉及实际的物理连接的网络技术提供载波状态信息；载波的存在意味着硬件功能是正常的。例如，以太网适配器能感知线路上的载波信号；当用户断开电缆时，载波消失，链路就不能正常工作了。默认情况下，网络设备假定存在载波信号。但是，利用下面的函数，驱动程序可显式改变这个状态：

```
void netif_carrier_off(struct net_device *dev);
void netif_carrier_on(struct net_device *dev);
```

如果驱动程序检测出设备上不存在载波，则应该调用 *netif_carrier_off* 通知内核这一情况。当载波再次出现时，应调用 *netif_carrier_on*。某些驱动程序在发生重要的配置变化时（比如介质类型），也会调用 *net_carrier_off*；一旦适配器完成了本身的重置，就会检测到新的载波，而数据传输可以重新开始。

还存在一个返回整型的函数：

```
int netif_carrier_ok(struct net_device *dev);
```

该函数用来检测当前的载波状态（和设备结构中反映的状态一样）。

套接字缓冲区

我们已经讨论了大部分和网络结构相关的问题，但尚未详细讲解 `sk_buff` 结构。该结构在Linux内核中处于网络子系统的核心地位，接下来介绍该结构的主要成员，以及用来操作这个结构的重要函数。

尽管没有严格要求读者理解 `sk_buff` 的必要，但是在跟踪问题或者试图优化代码时，如果能够看懂该结构的内容，则会很有帮助。例如，在 *loopback.c* 中，会发现作者在理解 `sk_buff` 内部细节的基础上对代码进行了优化。但也要注意如下警告：如果编写的代码使用了 `sk_buff` 结构的内部细节，则可能会在未来内核发布时出现问题。当然，有时性能上的好处和额外的维护成本是成正比的。

在这里不会描述整个结构，只会描述驱动程序可能会用到的那些成员。如果读者想了解更多的成员，可参考 *<linux/skbuff.h>* 文件，其中定义了这个结构以及操作该结构的函数原型。其他成员和函数用法的相关信息，也可通过 grep 内核源代码而获得。

重要的成员

这里介绍的成员是驱动程序可能会访问到的成员，它们与排列顺序无关。

```
struct net_device *dev;
```

接收和发送该缓冲区的设备。

```
union { /* ... */ } h;
union { /* ... */ } nh;
union { /*... */} mac;
```

指向数据包中各个层的报文头的指针。联合中的每个成员是指向不同数据结构类型的指针。h 中包含有传输层报文头（例如 `struct tcphdr *th`）的指针；nh 包含网络层报文头（比如 `struct iphdr *iph`）的指针；而 mac 中包含的是链路层报文头（例如 `struct ethdr *ethernet`）的指针。

如果驱动程序需要查询 TCP 数据包的源和目标地址，可在 `skb->h.th` 中找到这些地址。头文件中定义了可通过这种方式访问的完整报文头类型。

注意网络驱动程序要负责设置传入数据包的 mac 指针。这个任务通常由 *ether_type_trans* 函数处理，但是非以太网驱动程序需要直接设置 `skb->mac.raw`，将在“非以太网报文头”一节中说明。

```
unsigned char *head;
unsigned char *data;
unsigned char *tail;
unsigned char *end;
```

指向数据包中数据的指针。head 指向已分配空间的开头，data 是有效 octet（通常要比 head 大一些）的开头，tail 是有效 octet 的结尾，而 end 指向 tail 可达到的最大地址。另外还可以从这些指针得出可用缓冲区空间为 `skb->end - skb->head`，而当前已使用的数据空间为 `skb->tail - skb->data`。

```
unsigned int len;
unsigned int data_len;
```

len 是在数据包中全部数据的长度，data_len 是分隔存储的数据片段的长度。如果使用 scatter/gather I/O，则设置 data_len 为 0。

```
unsigned char ip_summed;
```

对数据包的校验策略。该成员由驱动程序对传入数据包进行设置，在“数据包的接收”一节中有过讨论。

```
unsigned char pkt_type;
```

在发送过程中使用的数据包类型。驱动程序负责将其设置为PACKET_HOST(该数据包是给我的)、PACKET_OTHERHOST(不，该数据包不是我的)、PACKET_BROADCAST或者是PACKET_MULTICAST。以太网驱动程序不必显式修改pkt_type，因为*eth_type_trans*会完成这个工作。

```
shinfo(struct sk_buff *skb);
unsigned int shinfo(skb)->nr_frags;
skb_frag_t shinfo(skb)->frags;
```

出于对性能方面的考虑，skb中的一些信息分散存储在内存中与其紧邻的其他结构中。该“共享信息”(因为它共享于网络代码中的各个skb拷贝中，因此这么称呼它)只能通过*shinfo*宏来访问。在该结构中有许多成员，但是它们中的大多数已经超过了本章所讨论的范围。本书只讨论nr_frags和在“分散/聚集I/O”一节中出现的标志。

本书不是特别关注结构中的剩余成员。它们用来维护缓冲区链表，记录拥有该缓冲区的套接字所占用的内存，等等。

操作套接字缓冲区的函数

使用sk_buff结构的网络设备通过一些正式的接口函数来操作该结构。操作套接字缓冲区的函数很多，下面是其中最重要的一些函数：

```
struct sk_buff *alloc_skb(unsigned int len, int priority);
struct sk_buff *dev_alloc_skb(unsigned int len);
```

分配一个缓冲区。*alloc_skb*函数分配一个缓冲区并初始化skb->data和skb->tail为skb->head。*dev_alloc_skb*函数以GFP_ATOMIC优先级调用*alloc_skb*，并在skb->head和skb->data之间保留一些空间，网络层使用这一数据空间进行优化工作，驱动程序不应访问这个空间。

```
void kfree_skb(struct sk_buff *skb);
void dev_kfree_skb(struct sk_buff *skb);
void dev_kfree_skb_irq(struct sk_buff *skb);
void dev_kfree_skb_any(struct sk_buff *skb);
```

释放一个缓冲区。*kfree_skb*函数由内核内部调用。驱动程序应该使用一种

*dev_kfree_skb*形式的函数，在非中断上下文中使用*dev_kfree_skb*，在中断上下文中使用*dev_kfree_skb_irq*，*dev_kfree_skb_any*函数可在上面两个上下文中使用。

```
unsigned char *skb_put(struct sk_buff *skb, int len);
unsigned char *__skb_put(struct sk_buff *skb, int len);
```

更新sk_buff结构中的tail和len成员；可用这些函数在缓冲区尾部添加数据。每个函数的返回值是skb->tail的先前值（换句话说，它指向刚刚建立的数据空间）。驱动程序可以使用该返回值，并通过调用memcpy(skb_put(...), data, len)来拷贝数据。这两个函数之间不同的是，*skb_put*会检查放入缓冲区的数据，而*__skb_put*忽略这个检查过程。

```
unsigned char *skb_push(struct sk_buff *skb, int len);
unsigned char *__skb_push(struct sk_buff *skb, int len);
```

上述函数减少skb->data，并增加skb->len。除了数据添加在数据包的头部而不是尾部之外，其功能类似*skb_put*。返回值指向刚刚创建的数据空间。在传输数据包之前，可使用该函数添加硬件头。和前面一样，*__skb_push*只在是否检查可用空间上和*skb_push*不同。

```
int skb_tailroom(struct sk_buff *skb);
```

该函数返回缓冲区中可用空间的大小。如果驱动程序在缓冲区中放入多于其能容纳的数据，则系统会出现panic。尽管读者可能会反对说*printk*足够标记该错误了，但内存破坏对系统非常有害，因此内核开发人员决定在这种情况下采取最后的决定性动作，即panic。实际情况下，如果正确分配了缓冲区，就不需要检查可用空间。因为驱动程序通常可在分配缓冲区之前获得数据包的大小，因此，只有被严重破坏的驱动程序才会向缓冲区中放入太多数据，这时，作为惩罚，内核就会出现panic。

```
int skb_headroom(struct sk_buff *skb);
```

返回在data之前可用的空间的数量，也即有多少octet能够保存在缓冲区。

```
void skb_reserve(struct sk_buff *skb, int len);
```

这个函数增加data和tail。该函数可在填充缓冲区之前保留报文头空间（headroom）。大多数以太网接口在数据包之前保留2个字节，这样IP头可在14字节的以太网头之后，在16字节边界上对齐。*snull*也这样做了。在“数据包的接收”一节中，为了避免引入额外的概念，我们没有提及这一点。

```
unsigned char *skb_pull(struct sk_buff *skb, int len);
```

从数据包头中删除数据。驱动程序无需使用这个函数，包含该函数只是为了完整性的需要。它减少skb->len并增加skb->data；这是从传入数据包的头部剥离硬件头（以太网或等价硬件）所使用的方法。

```
int skb_is_nonlinear(struct sk_buff *skb);
```

如果为使用分散/聚集I/O而将skb分解为多个数据片段，则返回真值。

```
int skb_headlen(struct sk_buff *skb);
```

返回skb第一个段的长度（该部分指向了skb->data）。

```
void *kmap_skb_frag(skb_frag_t *frag);
void kunmap_skb_frag(void *vaddr);
```

如果要在内核中直接访问非线性skb中的数据片段，这些函数负责映射和解除映射。由于使用了原子的kmap，因此不能同时映射多个数据片段。

内核定义了其他一些操作套接字缓冲区的函数，但这些函数主要用于上层网络代码，驱动程序不需要这些函数。

MAC 地址解析

以太网通信中有一个有趣的问题，即如何将IP号和MAC地址（接口的唯一硬件ID）关联起来。大部分协议也有类似的问题，但在这里重点讲述类以太网的情形。本节将提供有关该问题的完整描述，其中涉及到三种情形：ARP、无ARP的以太网头（类似*plip*），以及非以太网头。

在以太网中使用 ARP

通常处理地址解析的方法是使用ARP，即地址解析协议。幸运的是，ARP由内核维护，而以太网接口不需要做任何特殊工作就能支持ARP。只要在打开时正确设置dev->addr和dev->addr_len，驱动程序就无需担心将IP号解析为MAC地址这件事；*ether_setup*将把正确的设备函数赋予dev->hard_header和dev->rebuild_header。

尽管通常是由内核处理地址解析的细节（并缓存其结果），但它要调用接口的驱动程序来帮助建立数据包。毕竟驱动程序了解物理层数据包头的细节，而网络代码的作者试图将内核的其余部分和ARP隔离开来。为此，内核调用驱动程序的*hard_header*函数，将ARP查询的结果安排在数据包的适当位置。通常以太网驱动程序编写者无需了解这个过程——通用的以太网代码会处理这一切。

重载 ARP

类似*plip*的简单点对点网络接口可从以太网头中获得一些信息，但可避免因来回发送ARP数据包而带来的开销。*snull*中的示例代码也属于这种网络设备类型。*snull*不能使

用ARP，这是因为驱动程序修改了正在传输的数据包中的IP地址，而同时ARP数据包也修改了IP地址。尽管可以实现一个简单的ARP应答生成器，但解释直接处理物理层数据包头的方法，要更为直观一些。

如果设备希望使用常用的硬件头，而不运行ARP，则需要重载默认的*dev->hard_header*函数。下面是*snull*对该函数的实现，代码很短：

```
int snull_header(struct sk_buff *skb, struct net_device *dev,
                 unsigned short type, void *daddr, void *saddr,
                 unsigned int len)
{
    struct ethhdr *eth = (struct ethhdr *)skb_push(skb,ETH_HLEN);

    eth->h_proto = htons(type);
    memcpy(eth->h_source, saddr ? saddr : dev->dev_addr, dev->addr_len);
    memcpy(eth->h_dest,   daddr ? daddr : dev->dev_addr, dev->addr_len);
    eth->h_dest[ETH_ALEN-1]   ^= 0x01; /* dest是当前值异或1的结果 */
    return (dev->hard_header_len);
}
```

这个函数根据内核提供的信息，将其格式化成一个标准的以太网头。它同时切换了目标以太网地址的一个位，其原因将在后面介绍。

在接口接收到一个数据包时，*eth_type_trans*会以两种方式使用硬件头。我们已经在*snull_rx*中看到过这个函数：

```
skb->protocol = eth_type_trans(skb, dev);
```

该函数从以太网头中获得协议标识符（这里是ETH_P_IP）；它还对skb->mac.raw进行赋值，从数据包数据中删除硬件头（使用*skb_pull*），并设置skb->pkt_type。skb->pkt_type在分配skb时赋予其默认值为PACKET_HOST（表示这个数据包是发送到该主机的），而*eth_type_trans*会根据以太网的目标地址修改它。如果目标地址和接收它的接口地址不匹配，会将pkt_type设置为PACKET_OTHERHOST。接下来除非该接口处于混杂模式，或者在内核中设置了数据包转发，*netif_rx*会丢弃所有类型为PACKET_OTHERHOST的数据包。为此，*snull_header*小心处理，以保证目标硬件地址和“接收”接口的地址的匹配。

如果接口是点对点链路，则不希望接收非预期的组播数据包。为了避免这个问题，要记住第一个octet的最低位（LSB）为0的目标地址是发送到单个主机的（也就是说，它要么是PACKET_HOST，要么是PACKET_OTHERHOST）。*plip*驱动程序使用0xfc作为其硬件地址的第一个octet，而*snull*使用0x00。这两个地址均可以在类以太网的点对点链路中工作。

非以太网头

刚刚讨论了硬件头中除目标地址之外，还包含其他一些信息，其中最重要的是通信协议。现在介绍硬件头是如何封装相关信息的。如果读者需要了解细节，可从内核源代码或特定传输介质的技术文献中找到。大多数驱动程序编写者可忽略这个小节，而仅仅使用以太网的实现。

值得注意的是，并不是所有的信息都能为各个协议所提供。类似*plip*的点对点链路或者*snull*可在不减少一般性的情况下避免传输整个以太网头。前面描述过的由*snull_header*实现的*hard_header*设备函数接收从内核传递出来的信息 —— 协议层和硬件地址。它还从type参数中接收16位的协议号；例如IP协议由ETH_P_IP标识。驱动程序希望能够向接收主机正确地递交数据包数据以及协议号。点对点链路可在硬件头中省略地址，而仅仅传输协议号，这是因为数据包的发送和源及目标地址无关。一个仅仅传输IP数据包的链路，甚至可以避免传输任何的硬件头。

当链路的另一端接收到数据包时，驱动程序的接收函数应该正确设置skb->protocol、skb->pkt_type和skb_mac.raw 成员。

skb->mac.raw是一个字符指针，由上层网络代码（例如*net/ipv4/arp.c*）实现的地址解析机制使用。它必须指向与dev->type匹配的一个机器地址。设备类型的可能值定义在<*linux/if_arp.h*>中；以太网接口使用ARPHRD_ETHER。作为示例，下面是*eth_type_trans*处理已接收数据包的以太网头的代码：

```
skb->mac.raw = skb->data;
skb_pull(skb, dev->hard_header_len);
```

在最简单的情况下（没有头的点对点链路），skb->mac.raw可指向一个静态的缓冲区，其中包含接口的硬件地址，而protocol可被设置为ETH_P_IP，packet_type可保留其默认值，即PACKET_HOST。

因为每个硬件类型都是唯一的，因此很难给出比上述讨论更多的建议。但内核中充满了实际的例子。例如可以参考AppleTalk驱动程序（*drivers/net/appletalk/cops.c*），红外驱动程序（*drivers/net/irda/smc_ircc.c*），或者PPP驱动程序（*drivers/net/ppp_generic.c*）。

定制 ioctl 命令

读者已经知道了在套接字上实现的*ioctl*系统调用；SIOCSIFADDR和SIOCSIFMAP是“套接字*ioctl*”的两个例子。现在讨论网络代码如何使用该系统调用的第三个参数。

当为某个套接字使用*ioctl*系统调用时，命令号是定义在*<linux/sockios.h>*中的某个符号，而函数*sock_ioctl*直接调用一个协议相关的函数（这里的“协议”指主要的网络协议，例如IP或AppleTalk）。

任何协议层不能识别的*ioctl*命令都会传递到设备层。这些设备相关的*ioctl*命令从用户空间接受第三个参数，即一个`struct ifreq *`指针。这个结构在*<linux/if.h>*中定义。`SIOCSIFADDR`和`SIOCSIFMAP`命令实际使用了`ifreq`结构。`SIOCSIFMAP`的其他参数，尽管被定义为`ifmap`，但其实只是`ifreq`的一个成员。

除了使用标准化调用之外，每个接口可以定义它自己的*ioctl*命令。例如*plip*接口允许通过*ioctl*修改接口内部的超时值。套接字的*ioctl*实现能够识别16个接口私有的命令：从`SIOCDEVPRIVATE`到`SIOCDEVPRIVATE+15`（注2）。

如果识别出是上述命令之一，则在相关接口驱动程序中调用`dev->do_ioctl`。该函数接收相同的`struct ifreq *`指针，其与常用*ioctl*函数使用的一样。它的原型如下：

```
int (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);
```

`ifr`指针指向内核空间的地址，其中保存有用户传递结构的复本。在*do_ioctl*返回时，该结构将复制回用户空间；这样驱动程序可使用私有命令来接收和返回数据。

设备特有的命令可选择使用`struct ifreq`中的成员，但是它们已经具有标准的含义，而且驱动程序也很难使这个结构适应自己的需求。`ifr_data`成员是一个`caddr_t`型数据（一个指针），可用来满足设备特有的需求。驱动程序和调用*ioctl*命令的程序，要在`ifr_data`的使用上达成一致。例如*pppstats*使用设备特有的命令从*ppp*接口驱动程序中检索信息。

在这里讲解*do_ioctl*的实现不太值得，但通过本章以及内核中的实例，读者应该能够编写一个满足自己需求的*do_ioctl*函数。但要注意，*plip*使用`ifr_data`的方法不正确，所以不应作为*ioctl*的实现样例。

统计信息

驱动程序需要的最后一个函数是*get_stats*。这个函数返回设备统计结构的指针。它的实现非常简单；下面给出的这个实现甚至能够用于同一驱动程序管理多个接口的情况，因为统计信息一般保存在设备的数据结构中。

注2：注意，根据*<linux/sockios.h>*，`SIOCDEVPRIVATE`命令已经被废弃。但是，取代该命令的命令尚不明晰，因此，大量的驱动程序仍在使用这些命令。

```
struct net_device_stats *snull_stats(struct net_device *dev)
{
    struct snull_priv *priv = netdev_priv(dev);
    return &priv->stats;
}
```

用来返回实际统计信息的代码分布在的驱动程序中各处，在那里许多成员都被更新了。下面的清单给出了net_device_stats结构中最有意思的一些成员：

```
unsigned long rx_packets;
unsigned long tx_packets;
```

接口成功传递的输入和输出数据包的总量。

```
unsigned long rx_bytes;
unsigned long tx_bytes;
```

接口接收和发送的字节总数。

```
unsigned long rx_errors;
unsigned long tx_errors;
```

错误接收和发送的个数。在数据包传输过程中，可能出现无数错误，因此net_device_stats结构中包含了6个接收错误计数器，以及5个发送错误计数器。完整的清单可见*<linux/netdevice.h>*。如果可能，驱动程序应该维护详细的错误统计，因为这对系统管理员跟踪问题非常有帮助。

```
unsigned long rx_dropped;
unsigned long tx_dropped;
```

在接收和发送过程中丢弃的数据包数目。在没有可用内存保存数据包数据时，会将数据包丢弃。tx_dropped很少用到。

```
unsigned long collisions;
```

因介质拥堵而导致的冲突个数。

```
unsigned long multicast;
```

接收到的组播数据包个数。

这里需要重复强调的是，即使在接口被关闭时，*get_stats*函数也可能会在任意时间被调用，因此只要*net_device*结构存在，驱动程序就必须保存统计信息。

组播

组播数据包是期望由多于一个主机、但不是所有主机接收的网络数据包。这一功能通过赋予针对一组主机的特殊硬件地址而完成。发送到其中一个特殊地址的数据包，应该由

该组中的所有主机接收到。对以太网而言，组播地址在目标地址的第一个octet的最低位设置为 1，而所有设备板卡将自己的硬件地址的相应位清零。

对主机分组和硬件地址的处理由应用程序和内核执行，而接口的驱动程序无需处理这些问题。

组播数据包的传输并不困难，因为它们看起来和其他数据包没有两样。接口在通信介质上传输这些数据包，但并不关心它们的目标地址。内核负责赋予一个正确的硬件目标地址；如果定义了 *hard_header* 设备函数，则不必查看内核准备的数据。

内核在任意给定时刻均要跟踪组播地址。组播的主机清单可能会频繁改变，因为这是可在任意给定时间内运行的应用程序的功能，而且是用户的选择。驱动程序应该负责接收感兴趣的组播地址清单，然后将发送到这些地址的任意数据包交付给内核。驱动程序实现组播清单的方法，在某种程度上依赖于底层硬件的工作方式。通常来说，考虑组播时，硬件可划分为三类：

- 不能处理组播的接口。这种接口要么接收直接发送到其硬件地址的数据包（包括广播数据包），要么接收所有数据包。它们只能通过接收所有数据包而接收组播的数据包，这样就可能因为大量“不感兴趣”的数据包而会使操作系统性能下降。通常不会将这类接口看成是能够进行组播的接口，因此，驱动程序不能在`dev->flags`中设置 `IFF_MULTICAST`。

 点对点接口是一种特殊情况，因为它们不会执行任何硬件过滤，而是接收所有的数据包。

- 能够区分组播数据包和其他数据包（主机到主机或者广播）的接口。可指示这类接口接收每个组播数据包，并且让软件确定主机是否为有效的接收者。这种情况下引入的开销是能够接受的，因为典型网络中的组播数据包数目比较少。

- 能够为组播地址进行硬件检测的接口。可传递给这类接口一个可接收的组播地址清单，然后接口将忽略其他的组播数据包。这对内核来讲是最好的情形，因为内核不需要在丢弃“不感兴趣的”数据包上浪费处理器时间。

因为第三种设备类型是最为常见的，所以内核不断利用这类高性能接口的能力。为此内核会在有效组播地址发生变化时通知驱动程序，并且将新的清单传递给驱动程序，这样接口就可以根据新的信息更新其硬件过滤器。

对组播的内核支持

对组播数据包的支持由如下几项组成：一个设备函数、一个数据结构以及若干设备标志。

`void (*dev->set_multicast_list)(struct net_device *dev);`

当与设备相关的机器地址清单发生变化时，调用这个设备函数。该函数还在dev->flags被修改时调用，因为某些标志（比如IFF_PROMISC）也需要对硬件过滤器进行重新编程。这个函数接收到一个指向net_device结构的指针并返回void。如果驱动程序不想实现这个方法的话，可设置这个成员为NULL。

`struct dev_mc_list *dev->mc_list;`

这是与设备关联的所有组播地址形成的一个链表。该结构的实际定义将在本节末尾讲述。

`int dev->mc_count;`

链表中的节点数目。这个信息有点冗余，但检查mc_count是否为零，是检查链表的一个便捷方法。

`IFF_MULTICAST`

除非驱动程序在dev->flags中设置这个标志，接口是不会请求处理组播数据包的。虽然如此，但当dev->flags发生变化时，内核会调用驱动程序的*set_multicast_list*函数。这是因为接口未被激活的时候，组播清单可能已经发生改变。

`IFF_ALLMULTI`

这个标志由网络软件在dev->flags中设置，用来告诉驱动程序检索来自网络的所有组播数据包。这在组播路由被使能时发生。如果设置了这个标志，dev->mc_list不应该用来过滤组播数据包。

`IFF_PROMISC`

当接口被设置为混杂模式时，在dev->flags中设置该标志。不管dev->mc_list中含有哪些主机地址，接口都应该接收所有的数据包。

驱动程序开发人员需要的最后一点知识是dev_mc_list结构的定义，该结构在 *<linux/netdevice.h>* 中定义：

```
struct dev_mc_list {
    struct dev_mc_list   *next;             /* 列表中的下一个地址 */
    __u8                 dmi_addr[MAX_ADDR_LEN]; /* 硬件地址 */
    unsigned char        dmi_addrlen;       /* 地址长度 */
    int                  dmi_users;         /* 用户的数量 */
    int                  dmi_gusers;        /* 组的数量 */
};
```

因为组播和硬件地址与实际的数据包传输是无关的，所以这个结构在各种网络实现上是可移植的，通过一个octet字符串和长度来标识每个地址，这点和dev->dev_addr类似。

一个典型实现

描述 *set_multicast_list* 的最好方法是使用一些伪代码。

下面的函数是一个功能完整的驱动程序对该函数的一个典型实现。这个驱动程序所控制的接口具有一个复杂的硬件数据包过滤器，它可以保存主机能接收的组播地址表。表的最大尺寸是 FF_TABLE_SIZE。

具有 ff_ 前缀的函数表示该函数是一个针对硬件的操作:

```
void ff_set_multicast_list(struct net_device *dev)
{
    struct dev_mc_list *mcptr;

    if (dev->flags & IFF_PROMISC) {
        ff_get_all_packets();
        return;
    }
    /* 如果不能处理完所有的地址，则获得所有组播数据包并且在软件中保存它们 */
    if (dev->flags & IFF_ALLMULTI || dev->mc_count > FF_TABLE_SIZE) {
        ff_get_all_multicast_packets();
        return;
    }
    /* 没有组播？只是获得自己的数据 */
    if (dev->mc_count = = 0) {
        ff_get_only_own_packets();
        return;
    }
    /* 在硬件过滤器中保存所有的组播地址 */
    ff_clear_mc_list();
    for (mc_ptr = dev->mc_list; mc_ptr; mc_ptr = mc_ptr->next)
        ff_store_mc_address(mc_ptr->dmi_addr);
    ff_get_packets_in_multicast_list();
}
```

如果接口不能在硬件过滤器中保存针对传入数据包的组播表，则可以简化这个实现。这种情况下，FF_TABLE_SIZE 减小为 0，也不需要最后四行代码。

正如前面提到，即使不能处理组播数据包的接口，也需要实现 *set_multicast_list* 方法，以便对 dev->flags 的变化做出响应。这种情况称之为“非完整”实现。其实现非常简单，代码如下所示:

```
void nf_set_multicast_list(struct net_device *dev)
{
    if (dev->flags & IFF_PROMISC)
        nf_get_all_packets();
    else
        nf_get_only_own_packets();
}
```

IFF_PROCMISC的实现非常重要，否则用户就无法运行*tcpdump*或其他网络分析器。另一方面，如果接口运行在点对点链路上，就根本没有必要实现*set_multicast_list*，因为用户总是会接收到所有的数据包。

其他知识点详解

本节将讲述一些网络驱动程序作者感兴趣的话题，其目的是让读者沿着正确的方向学习。如果要知道所有相关知识点的全貌，在内核源代码上花许多时间是必须的。

对介质无关接口的支持

介质无关接口（Media Independent Interface，MII）是一个IEEE802.3标准，它描述了以太网收发器是如何与网络控制器连接的。在市场上销售的大量产品都有这样的接口。如果要为MII控制器写一个驱动程序，仔细学习内核中的通用MII支持层，会使这个任务变得简单。

为了使用通用MII层，需要包含头文件*<linux/mii.h>*。无论是否使用全双工模式，都需要用收发器的物理ID填写mii_if_info结构。为使用mii_if_info结构，还需要使用两个函数：

```
int (*mdio_read) (struct net_device *dev, int phy_id, int location);
void (*mdio_write) (struct net_device *dev, int phy_id, int location, int val);
```

正如读者期望的那样，这两个函数实现了与特定MII接口的通信。

通用MII代码还提供了一套用于查询和修改收发器操作模式的函数，其中的驱动函数都被设计成与*ethtool*工具（在下一节讲述）配合使用。参看*<linux/mii.h>*和*drivers/net/mii.c*以了解实现细节。

ethtool支持

*ethtool*是为系统管理员提供的用于控制网络接口的工具。只有当驱动程序支持*ethtool*时，使用*ethtool*才能控制包括速度、介质类型、双工操作、DMA设置、硬件校验、LAN唤醒操作在内的许多接口参数。读者可以从站点*http://sf.net/projects/gkernel/*下载*ethtool*。

在*<linux/ethtool.h>*中含有为支持*ethtool*的相关声明。其核心是一个ethtool_ops类型的结构，它包含了*ethtool*支持的全部24个函数。这些函数中的大多数是相互关联的，

请参看 *<linux/ethtool.h>* 了解其细节。如果驱动程序使用了 MII 层，可以使用 *mii_ethtool_gset* 和 *mii_ethtool_sset* 以分别实现 *get_settings* 和 *set_settings* 函数。

为了使*ethtool*能与设备配合工作，必须在net_device结构中设置指向ethtool_ops结构的指针。因此可以使用宏SET_ETHTOOL_OPS（在*<linux/netdevice.h>*中定义）完成这一任务。请注意即使已经关闭了接口，*ethtool* 函数依然能够被调用。

Netpoll

"Netpoel" 是相当晚（2.6.5）才出现在网络栈中的；它出现的目的是让内核在网络和 I/O 子系统尚不能完整可用时，依然能发送和接收数据包。其用于网络控制台和远程内核调试中。无论如何，在驱动程序中支持netpoll不是必需的，但是它却能让驱动程序在某些时候特别有用。在许多情况下，对netpoll的支持是相对容易的。

实现netpoll的驱动程序需要实现*poll_controller*函数。该函数的作用是在缺少设备中断时，还能对控制器做出响应。几乎所有的 *poll_controller* 函数都有如下的形式：

```
void my_poll_controller(struct net_device *dev)
{
    disable_device_interrupts(dev);
    call_interrupt_handler(dev->irq, dev, NULL);
    reenable_device_interrupts(dev);
}
```

从根本上讲，*poll_controller* 函数只是模拟了来自指定设备的中断。

快速参考

这个小节给出了本章介绍过的概念的快速参考，同时解释了驱动程序应该包含的每个头文件。但是 net_device 和 sk_buff 结构的成员不会在这里重复。

```
#include <linux/netdevice.h>
```

这个头文件保存有net_device和net_device_stats结构的定义，并包含了网络驱动程序需要的其他几个头文件。

```
struct net_device *alloc_netdev(int sizeof_priv, char *name, void
  (*setup)(struct net_device *);
struct net_device *alloc_etherdev(int sizeof_priv);
void free_netdev(struct net_device *dev);
```

分配和释放 net_device 结构的函数。

```
int register_netdev(struct net_device *dev);
void unregister_netdev(struct net_device *dev);
```

注册和注销一个网络设备。

```
void *netdev_priv(struct net_device *dev);
```

获得指向网络设备结构中驱动程序私有数据区指针的函数。

```
struct net_device_stats;
```

保存设备统计信息的结构。

```
netif_start_queue(struct net_device *dev);
netif_stop_queue(struct net_device *dev);
netif_wake_queue(struct net_device *dev);
```

上述函数控制外发数据包向驱动程序的传递。在调用*netif_start_queue*之前，不会传输任何数据包。*netif_stop_queue*暂停传输，而*netif_wake_queue*重新启动队列并通知网络层重新启动数据包的传输。

```
skb_shinfo(struct sk_buff *skb);
```

提供对数据包缓存区中“共享信息”访问的宏。

```
void netif_rx(struct sk_buff *skb);
```

调用（包括中断期间）这个函数可通知内核已经接收到一个数据包，并封装入一个套接字缓冲区。

```
void netif_rx_schedule(dev);
```

调用该函数通知内核数据包已经存在，并且在接口上启动轮询机制；它只在NAPI驱动程序中使用。

```
int netif_receive_skb(struct sk_buff *skb);
void netif_rx_complete(struct net_device *dev);
```

这两个函数只在NAPI驱动程序中使用。NAPI中的*netif_receive_skb*函数与*netif_rx*等价；它将数据包发送给内核。当NAPI驱动程序耗尽了为接收数据包准备的内存，则它将重新启动中断，然后调用*netif_rx_complete*终止轮询函数。

```
#include <linux/if.h>
```

*netdevice.h*中包含该头文件。在该文件中声明了接口标志（IFF_ macros）和ifmap结构，在网络驱动程序的*ioctl*实现中，其扮演了重要角色。

```
void netif_carrier_off(struct net_device *dev);
void netif_carrier_on(struct net_device *dev);
int netif_carrier_ok(struct net_device *dev);
```

前两个函数告诉内核在指定接口上是否存在载波信号。*netif_carrier_ok*检查载波状态作为在device结构中的应答。

```
#include <linux/if_ether.h>
ETH_ALEN
ETH_P_IP
struct ethhdr;
```

*netdevice.h*中包含该头文件。*if_ether.h*中定义了所有的ETH_ 宏，用来表示octet的长度（比如地址长度）和网络协议（比如IP）。它还定义了ethhdr结构。

```
#include <linux/skbuff.h>
```

定义了sk_buff及其相关结构，同时定义了许多作用于缓冲区的内联函数。该头文件包含在*netdevice.h*中。

```
struct sk_buff *alloc_skb(unsigned int len, int priority);
struct sk_buff *dev_alloc_skb(unsigned int len);
void kfree_skb(struct sk_buff *skb);
void dev_kfree_skb(struct sk_buff *skb);
void dev_kfree_skb_irq(struct sk_buff *skb);
void dev_kfree_skb_any(struct sk_buff *skb);
```

分配和释放套接字缓冲区的函数。因此驱动程序通常使用具有dev_前缀的变种。

```
unsigned char *skb_put(struct sk_buff *skb, int len);
unsigned char *__skb_put(struct sk_buff *skb, int len);
unsigned char *skb_push(struct sk_buff *skb, int len);
unsigned char *__skb_push(struct sk_buff *skb, int len);
```

将数据添加到skb的函数；*skb_put*将数据放在skb的末尾，而*skb_push*将数据放在开头。常用的版本还负责检查是否有足够的空间存放数据；而有双下划线前缀的版本不进行该项检查。

```
int skb_headroom(struct sk_buff *skb);
int skb_tailroom(struct sk_buff *skb);
void skb_reserve(struct sk_buff *skb, int len);
```

在skb中实现空间管理的函数。*skb_headroom*和*skb_tailroom*分别返回在skb的开头和结尾，还有多少空间可用。*skb_reserve*用于在skb开头部分保留空间，保留的空间必须为空。

```
unsigned char *skb_pull(struct sk_buff *skb, int len);
```

*skb_pull*通过调整内部指针而“删除”skb内的数据。

```
int skb_is_nonlinear(struct sk_buff *skb);
```

如果使用分散/聚集I/O，并且skb分离成多个数据片段，则该函数返回真实值。

```
int skb_headlen(struct sk_buff *skb);
```

返回 skb 中 skb->data 的第一个段的长度。

```
void *kmap_skb_frag(skb_frag_t *frag);
void kunmap_skb_frag(void *vaddr);
```

提供对非线性 skb 中的数据片段的直接访问。

```
#include <linux/etherdevice.h>
void ether_setup(struct net_device *dev);
```

为以太网驱动程序设置大部分通用设备方法的函数。它还设置了 dev->flags。如果设备名称的第一个字符为空，或者是空格的话，这个函数将把下一个可用的ethx名称赋给 dev->name。

```
unsigned short eth_type_trans(struct sk_buff *skb, struct net_device *dev);
```

当以太网接口接收到一个数据包时，调用该函数设置skb->pkt_type。返回值是保存在 skb->protocol 中的协议号。

```
#include <linux/sockios.h>
SIOCDEVPRIVATE
```

16 个 *ioctl* 命令中的第一个，每个驱动程序都能够出于自身的考虑实现它。在 *sockios.h* 中定义了所有的网络 *ioctl* 命令。

```
#include <linux/mii.h>
struct mii_if_info;
```

支持实现 MII 标准的设备驱动程序的声明和结构。

```
#include <linux/ethtool.h>
struct ethtool_ops;
```

让设备可使用 *ethtool* 工具的声明和结构。

第十八章

TTY 驱动程序



tty 设备的名称是从过去的电传打字机缩写而来，最初是指连接到 Unix 系统上的物理或者虚拟终端。随着时间的推移，当通过串行口能够建立起终端连接后，这个名字也用来指任何的串口设备。物理 tty 设备的例子有串口、USB 到串口的转换器，还有需要特殊处理才能正常工作的调制解调器（比如传统的 WinModem 类设备）等。tty 虚拟设备支持虚拟控制台，它能通过键盘及网络连接或者通过 xterm 会话登录到计算机上。

Linux tty 驱动程序的核心紧挨在标准字符设备驱动层之下，并提供了一系列的功能，作为接口被终端类型设备使用。内核负责控制通过 tty 设备的数据流，并且格式化这些数据。这使得 tty 驱动程序把重点放在处理流向或者流出硬件的数据上，而不必重点考虑使用常规方法与用户空间的交互。为了控制数据流，有许多不同的线路规程（line discipline）可虚拟地“插入”任何的 tty 设备上，这由不同的 tty 线路规程驱动程序实现。

在图 18-1 中，tty 核心从用户那里得到将被发往 tty 设备的数据，然后把数据发送给 tty 线路规程驱动程序，该驱动程序负责把数据传递给 tty 驱动程序。tty 驱动程序对数据进行格式化，然后才能发送给硬件。从 tty 硬件那里接收的数据将回溯至 tty 驱动程序，然后流入 tty 线路规程驱动程序，接着是 tty 核心，最后用户从 tty 核心那里得到数据。有时 tty 驱动程序直接与 tty 核心通信，tty 核心将数据直接发送给 tty 驱动程序，但通常是 tty 线路规程驱动程序修改在二者之间流动的数据。

tty 线路规程对于 tty 驱动程序来说是不透明的。驱动程序不能直接与线路规程通信，甚至不知道它的存在。在某种意义上讲，驱动程序的作用是将发送给它的数据格式化成硬件能理解的格式，并从硬件那里接收数据。tty 线路规程的作用是使用特殊的方法，把从用户或者硬件那里接收的数据格式化。这种格式化通常使用一些协议来完成转换，比如 PPP 或者是蓝牙（Bluetooth）。

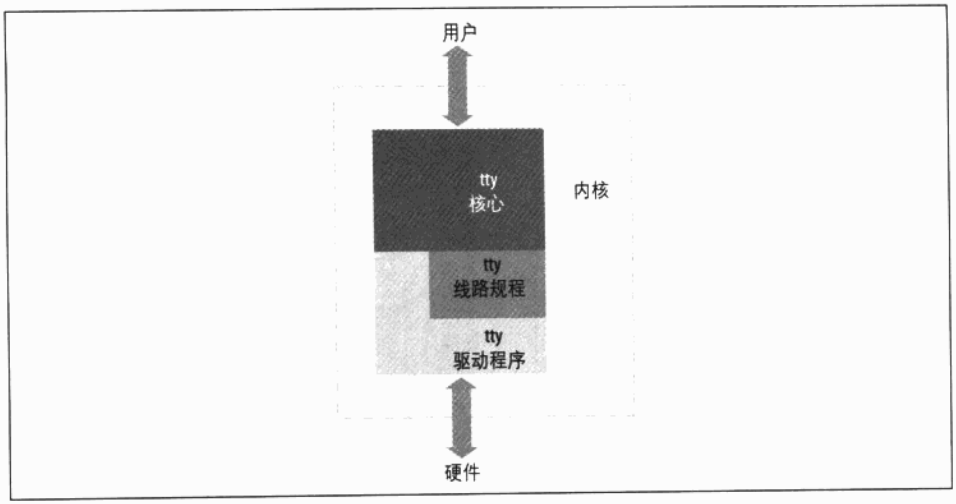


图 18-1：tty 核心概览

有三种类型的 tty 驱动程序：控制台、串口和 pty。控制台和 pty 驱动程序已经被编写好了，而且可能也不必为这两类 tty 驱动程序编写其他的驱动程序。这使得任何使用 tty 核心与用户和系统交互的新驱动程序都可被看成是串口驱动程序。

为了确定目前装载到内核中的是何种类型的 tty 驱动程序，并确定目前使用的是何种 tty 设备，我们可查阅 */proc/tty/drivers* 文件。该文件列举了当前使用的不同的 tty 驱动程序，显示了驱动程序的名称、默认的节点名称、驱动程序的主设备号、驱动程序所使用的次设备号范围以及 tty 驱动程序的类型。下面是该文件的一个例子：

```
/dev/tty             /dev/tty        5       0 system:/dev/tty
/dev/console         /dev/console    5       1 system:console
/dev/ptmx            /dev/ptmx       5       2 system
/dev/vc/0            /dev/vc/0       4       0 system:vtmaster
usbserial            /dev/ttyUSB   188   0-254 serial
serial               /dev/ttyS       4   64-67 serial
pty_slave            /dev/pts      136   0-255 pty:slave
pty_master           /dev/ptm      128   0-255 pty:master
pty_slave            /dev/ttyp       3   0-255 pty:slave
pty_master           /dev/pty        2   0-255 pty:master
unknown              /dev/tty        4    1-63 console
```

如果 tty 驱动程序执行了所包含的功能，则 */proc/tty/driver/* 目录下包含了若干独立文件为 tty 驱动程序所使用。默认的串口驱动程序在该目录下创建了一个文件，显示了许多关于串行硬件的特殊信息。在该目录下如何创建文件，将在本章后面论述。

当前注册并存在于内核的 tty 设备在 */sys/class/tty* 下都有自己的子目录。在这些子目录

中，有一个“dev”文件包含了分配给该tty设备的主设备号和次设备号。如果驱动程序告诉内核物理设备的路径和分配给该tty设备的驱动程序，它将创建一个指向它们的符号连接。下面是该目录树的结构示例：

```
/sys/class/tty/
|-- console
|   `-- dev
|-- ptmx
|   `-- dev
|-- tty
|   `-- dev
|-- tty0
|   `-- dev
   ...
|-- ttyS1
|   `-- dev
|-- ttyS2
|   `-- dev
|-- ttyS3
|   `-- dev
   ...
|-- ttyUSB0
|   |-- dev
|   |-- device -> ../../../devices/pci0000:00/0000:00:09.0/usb3/3-1/3-
1:1.0/ttyUSB0
|   `-- driver -> ../../../bus/usb-serial/drivers/keyspan_4
|-- ttyUSB1
|   |-- dev
|   |-- device -> ../../../devices/pci0000:00/0000:00:09.0/usb3/3-1/3-
1:1.0/ttyUSB1
|   `-- driver -> ../../../bus/usb-serial/drivers/keyspan_4
|-- ttyUSB2
|   |-- dev
|   |-- device -> ../../../devices/pci0000:00/0000:00:09.0/usb3/3-1/3-
1:1.0/ttyUSB2
|   `-- driver -> ../../../bus/usb-serial/drivers/keyspan_4
`-- ttyUSB3
    |-- dev
    |-- device -> ../../../devices/pci0000:00/0000:00:09.0/usb3/3-1/3-
1:1.0/ttyUSB3
    `-- driver -> ../../../bus/usb-serial/drivers/keyspan_4
```

小型 TTY 驱动程序

为了说明tty核心是如何工作的，首先创建一个可以被加载的小型tty驱动程序，并对其进行读写操作，然后卸载它。任何tty驱动程序的主要数据结构是结构tty_driver。它被用来向tty核心注册和注销驱动程序，对其的描述包含在内核头文件 *<linux/tty_driver.h>* 中。

为了创建tty_driver结构的对象，必须把该驱动所支持的tty设备的数量作为参数调用函数*alloc_tty_driver*。下面的一小段代码可以执行这个操作：

```
/* 分配tty驱动程序 */
tiny_tty_driver = alloc_tty_driver(TINY_TTY_MINORS);
if (!tiny_tty_driver)
    return -ENOMEM;
```

当函数*alloc_tty_driver*被成功调用后，tty_driver结构将根据tty驱动程序的需求，用正确的信息初始化。该结构体包含了大量的成员，但为使tty驱动程序能正常工作，并不是所有的成员都需要被初始化。下面的例子说明如何初始化该结构，以及如何设置足够多的成员来创建tty驱动程序。它使用*tty_set_operations*函数拷贝了在驱动程序中定义的一系列操作函数：

```
static struct tty_operations serial_ops = {
    .open = tiny_open,
    .close = tiny_close,
    .write = tiny_write,
    .write_room = tiny_write_room,
    .set_termios = tiny_set_termios,
};

...

    /* 初始化tty驱动程序 */
    tiny_tty_driver->owner = THIS_MODULE;
    tiny_tty_driver->driver_name = "tiny_tty";
    tiny_tty_driver->name = "ttty";
    tiny_tty_driver->devfs_name = "tts/ttty%d";
    tiny_tty_driver->major = TINY_TTY_MAJOR,
    tiny_tty_driver->type = TTY_DRIVER_TYPE_SERIAL,
    tiny_tty_driver->subtype = SERIAL_TYPE_NORMAL,
    tiny_tty_driver->flags = TTY_DRIVER_REAL_RAW | TTY_DRIVER_NO_DEVFS,
    tiny_tty_driver->init_termios = tty_std_termios;
    tiny_tty_driver->init_termios.c_cflag = B9600 | CS8 | CREAD | HUPCL | CLOCAL;
    tty_set_operations(tiny_tty_driver, &serial_ops);
```

上面列举的变量和函数，以及如何使用这个结构体，将在本章余下的部分说明。

为了向tty核心注册这个驱动程序，必须将tty_driver结构传递给*tty_register_driver*函数：

```
/* 注册tty驱动程序 */
retval = tty_register_driver(tiny_tty_driver);
if (retval) {
    printk(KERN_ERR "failed to register tiny tty driver");
    put_tty_driver(tiny_tty_driver);
    return retval;
}
```

当*tty_register_driver*被调用时，内核将根据tty驱动程序所拥有的所有次设备号，创建所有的不同sysfs tty文件。如果使用了*devfs*（本书没有讲述），并且不设置TTY_DRIVER_NO_DEVFS标志位的情况下，也将创建devfs文件。如果要让用户看到系统中已真实存在的设备，并且为用户保持更新，*则可以调用tty_register_device*，并设置该标志位，而这正是*devfs*用户所期望的。

在注册自身后，驱动程序使用*tty_register_device*函数注册它所控制的设备。该函数有三个参数：

- 属于该设备的tty_driver结构指针。
- 设备的次设备号。
- 指向该tty设备所绑定的device结构指针。如果tty设备没有绑定任何device结构，该参数为NULL。

因为我们的设备是虚拟的，并且不和任何物理设备绑定，因此，我们的驱动程序将立刻注册所有的tty设备：

```
for (i = 0; i < TINY_TTY_MINORS; ++i)
    tty_register_device(tiny_tty_driver, i, NULL);
```

为了向tty核心注销驱动程序，所有使用*tty_register_device*函数注册的tty设备都需要使用*tty_unregister_device*函数清除自身。然后，必须调用*tty_unregister_driver*注销tty_driver结构。

```
for (i = 0; i < TINY_TTY_MINORS; ++i)
    tty_unregister_device(tiny_tty_driver, i);
tty_unregister_driver(tiny_tty_driver);
```

termios结构

在结构tty_driver中的init_termios变量是一个termios结构。如果用户在端口初始化以前就使用了该端口，那么该变量用来提供一系列安全的设置值。驱动程序用一系列标准值初始化该变量，而这些值是从tty_std_termios值中拷贝过来的。tty_std_termios结构在tty核心中被定义如下：

```
struct termios tty_std_termios = {
    .c_iflag = ICRNL | IXON,
    .c_oflag = OPOST | ONLCR,
    .c_cflag = B38400 | CS8 | CREAD | HUPCL,
    .c_lflag = ISIG | ICANON | ECHO | ECHOE | ECHOK |
               ECHOCTL | ECHOKE | IEXTEN,
    .c_cc = INIT_C_CC
};
```

termios结构被用来为在tty设备上的某个特定端口保存当前所有设置。这些设置控制着当前的波特率、数据大小、数据流参数和其他一些值。该结构的各个成员的含义是:

tcflag_t c_iflag;
 输入模式标志

tcflag_t c_oflag;
 输出模式标志

tcflag_t c_cflag;
 控制模式标志

tcflag_t c_lflag;
 本地模式标志

cc_t c_line;
 线路规程类型

cc_t c_cc[NCCS];
 控制字符数组

所有的模式标志都在一个较大的位成员中定义。各种不同模式的值和它们的用途，可以在任何Linux发行版中关于termios的手册页中找到。内核提供了一套有用的宏来获得不同位的值。这些宏在头文件 *include/linux/tty.h* 中定义。

为使tty驱动程序能正常工作，在tiny_tty_driver变量中所有的字段都是必需的。需要owner成员是因为要避免当tty端口打开时，tty驱动程序被卸载。在以前的内核版本中，tty驱动程序本身负责处理模块引用计数逻辑。但是内核程序员发现：要解决全部可能的竞态问题是十分困难的，因此现在的tty核心为tty驱动程序处理所有的控制问题。

driver_name和name成员看起来十分相似，然而它们有不同的用途。在内核所有的tty驱动程序中，driver_name变量被设置成简短、描述性的、唯一的值。这是因为在 */proc/tty/drivers* 文件中要向用户描述驱动程序的情况，并且在sysfs的tty类目录中显示当前被加载的tty驱动程序。name成员是在 */dev* 目录中，定义分配给单独tty节点的名字。通过在该名字末尾添加tty设备序号来创建tty设备。它还被用来在sysfs的 */sys/class/tty* 目录中创建设备名。如果在内核中使用devfs，该名字可以包含放置tty驱动程序的任何子目录。举个例子，如果使用devfs，在内核中的串口驱动程序设置name成员为tts，如果不使用devfs，则是ttyS。该字符串也会出现在 */proc/tty/drivers* 文件中。

如同以前提到过的，*/proc/tty/drivers* 文件显示了当前注册的所有tty驱动程序。驱动程序在内核中使用 *tiny_tty* 进行注册，如果不使用 *devfs*，该文件与下面的例子类似:

```
$ cat /proc/tty/drivers
tiny_tty             /dev/ttty     240     0-3 serial
usbserial            /dev/ttyUSB   188   0-254 serial
serial               /dev/ttyS       4  64-107 serial
pty_slave            /dev/pts      136   0-255 pty:slave
pty_master           /dev/ptm      128   0-255 pty:master
pty_slave            /dev/ttyp       3   0-255 pty:slave
pty_master           /dev/pty        2   0-255 pty:master
unknown              /dev/vc/        4    1-63 console
/dev/vc/0            /dev/vc/0       4       0 system:vtmaster
/dev/ptmx            /dev/ptmx       5       2 system
/dev/console         /dev/console    5       1 system:console
/dev/tty             /dev/tty        5       0 system:/dev/tty
```

同样，当tiny_tty驱动程序向tty核心注册时，sysfs的目录*/sys/class/tty*也有如同下面的类结构：

```
$ tree /sys/class/tty/ttty*
/sys/class/tty/ttty0
`-- dev
/sys/class/tty/ttty1
`-- dev
/sys/class/tty/ttty2
`-- dev
/sys/class/tty/ttty3
`-- dev

$ cat /sys/class/tty/ttty0/dev
240:0
```

major变量表示该驱动程序使用的主设备号。type和subtype变量声明该驱动程序是什么类型的tty驱动程序。对我们的例子，这是一个“常规（normal）”类型的串口驱动程序。tty驱动程序另外一个唯一子类型是“呼出（callout）”类型。呼出设备传统上被用来控制设备的线路规程设置。数据的发送或者接收将通过某个设备节点进行，而任何对线路设置的改变将被发往不同的设备节点，这就是呼出设备。这需要为每个单独的tty设备使用两个次设备号。幸运的是几乎所有驱动设备都在同一个设备节点上处理数据和线路设置，在新的驱动程序中，呼出类型已经很少使用了。

tty驱动程序和tty核心都使用flags变量表明当前驱动程序的状态以及该tty驱动程序的类型。这里定义了许多位的掩码宏操作，当对这些标志位进行测试和操作时，必须使用这些宏。驱动程序可以设置flags变量中的三个位：

TTY_DRIVER_RESET_TERMIOS

该标志表示当最后一个进程关闭该设备时，tty核心对termios设置复位。这对控制台和pty驱动程序来说很有用。比如用户将终端设置为非常规状态，如果设置了该位，当用户退出或者控制该会话的进程被关闭时，终端能够自动恢复常规值。

TTY_DRIVER_REAL_RAW

该标志表示tty驱动程序使用奇偶校验或者中断字符线路规程。这使得线路规程能以比较快的方式接收字符，因为它不必检查从tty驱动程序那里接收的每一个字符。由于速度提升的好处，所有的tty驱动程序通常都设置该位。

TTY_DRIVER_NO_DEVFS

该位表示当调用 *tty_register_driver* 时，tty核心不需要为tty驱动程序创建任何的devfs入口。这对于那些需要动态创建和删除次设备的驱动程序来说非常有用。举几个设置该位的驱动程序的例子：比如USB到串口驱动程序，USB调制解调器驱动程序，USB蓝牙tty驱动程序，以及许多标准串口驱动程序。

当tty驱动程序要向tty核心注册一个特殊的tty设备时，它必须调用 *tty_register_device* 函数，并且把指向tty驱动程序的指针，还有它所创建设备的次设备号作为参数。如果不这么做，tty核心依然将所有调用传递给tty驱动程序，但是可能不提供一些内部与tty相关的操作。这将包括 */sbin/hotplug* 发出的新tty设备的通知消息，以及在sysfs中tty设备的表示。当把已经注册的tty设备从系统移除时，tty驱动程序必须调用 *tty_unregister_device* 函数。

在该变量中还有一位TTY_DRIVER_INSTALLED，它由tty核心控制。当tty驱动程序注册后，由tty核心而不是tty驱动程序置位。

tty_driver 函数指针

最后，*tiny_tty* 驱动程序声明了四个函数指针。

open 和 close

当用户使用 *open* 打开由驱动程序分配的设备节点时，tty核心将调用open函数。tty核心使用分配给该设备tty_struct结构的指针，以及一个文件描述符作为参数调用该函数。tty驱动程序一定要设置open成员才能正常工作，否则用户调用open时，将返回-ENODEV。

当调用 *open* 函数时，tty驱动程序或者将数据保存到传递给它的tty_struct变量中，或者将数据保存在一个静态数组中，然后通过分配给该端口的次设备号进行引用。该步骤是必须的，这样在以后调用close、write和其他函数时，tty驱动程序能够知道是对哪个设备进行操作的。

tiny_tty 驱动程序在tty结构中保存了一个指针，请参看下面的代码：

```
static int tiny_open(struct tty_struct *tty, struct file *file)
{
    struct tiny_serial *tiny;
    struct timer_list *timer;
    int index;

    /* 一旦发生错误，则初始化指针 */
    tty->driver_data = NULL;

    /* 获得与tty指针相关的串口对象 */
    index = tty->index;
    tiny = tiny_table[index];
    if (tiny == NULL) {
        /* 第一次访问该设备，创建它 */
        tiny = kmalloc(sizeof(*tiny), GFP_KERNEL);
        if (!tiny)
            return -ENOMEM;

        init_MUTEX(&tiny->sem);
        tiny->open_count = 0;
        tiny->timer = NULL;

        tiny_table[index] = tiny;
    }
    down(&tiny->sem);
    /* 在tty结构中保存上述结构 */

    tty->driver_data = tiny;
    tiny->tty = tty;
```

在这段代码中，tty结构中保存了tiny_serial结构。这使得*tiny_write*、*tiny_write_room*、*tiny_close*函数能够获得tiny_serial结构，并能正确处理它。

tiny_serial结构定义如下：

```
struct tiny_serial {
    struct tty_struct   *tty;        /* 指向该设备的tty指针 */
    int                 open_count;  /* 该端口被打开的次数 */
    struct semaphore    sem;         /* 锁住该结构 */
    struct timer_list   *timer;
};
```

如上所示，当端口第一次被打开，在调用open函数时open_count变量被初始化为0。这是一个典型的引用计数器，由于对同一个设备，为了能使多个进程进行读写数据，tty驱动程序的*open*和*close*函数会被多次调用，因此该参数是需要的。为了能正确处理好每一件事，端口被打开和关闭的次数必须被记录；端口被使用时，驱动程序会增减该次数的引用。当端口第一次被打开时，可以对所需要的任何硬件做初始化及对所需内存进行分配。当端口最后一次被关闭时，可以关闭任何需要的硬件，并对内存进行清理。

*tiny_open*函数的剩余部分显示了如何保存设备被打开的次数：

```
    ++tiny->open_count;
    if (tiny->open_count =  = 1) {
        /* 这是该端口第一次被打开 */
        /* 在这里做所需要的任何初始化工作 */
```

如果打开过程中出现意外导致操作失败，*open* 函数将返回一个负值错误号，而返回0则表示操作成功。

当用户使用先前由 *open* 函数创建的文件句柄作为参数调用 *close* 函数时，tty 核心调用 *close* 函数指针。此时设备将被关闭。然而，由于 *open* 函数可以被多次调用，*close* 函数也可以被多次调用。因此该函数应该能够记录它被调用的次数，这样就可以判断在本次调用时，是否要真的关闭硬件。*tiny_tty* 驱动程序使用下面的代码做到这一点：

```
static void do_close(struct tiny_serial *tiny)
{
    down(&tiny->sem);

    if (!tiny->open_count) {
        /* 端口从未被打开 */
        goto exit;
    }

    --tiny->open_count;
    if (tiny->open_count <= 0) {
        /* 最后一个用户已经将端口关闭 */
        /* 在这里做任何硬件相关操作 */

        /* 关闭定时器 */
        del_timer(tiny->timer);
    }
exit:
    up(&tiny->sem);
}

static void tiny_close(struct tty_struct *tty, struct file *file)
{
    struct tiny_serial *tiny = tty->driver_data;
    if (tiny)
        do_close(tiny);
}
```

为了关闭设备，*tiny_close* 函数只是调用了 *do_close* 函数来做这个工作。因此当端口被打开，而驱动程序被卸载时，不必在这里复制关闭设备的逻辑。*close* 函数没有返回值，所以也无法判断操作的成功与失败。

数据流

当数据要被发送给硬件时，用户调用 *write* 函数。首先tty核心接收到了该调用，然后内

核将数据发送给tty驱动程序的*write*函数。tty核心同时也告诉tty驱动程序所发送数据的大小。

有时由于tty硬件的速度及缓冲区大小的原因，当*write*函数被调用时，写操作程序所处理的数据不能同时都发送出去。*write*函数将返回发送给硬件的字符数(或者是最后一次被发送的队列)，因此用户程序可以检查该值以判断是否写入了所有的数据。与在内核驱动程序中等待所有数据都被发送出去相比，在用户空间做这件事更容易一些。在*write*调用时如果发生了任何错误，将返回一个为负值的错误号，而不是被写入的字符数。

可以在中断上下文或者是用户上下文中调用*write*函数。tty驱动程序在中断上下文中时，它不会调用任何可能休眠的函数，这点非常重要。这包含任何可能调用*schedule*的函数，比如常用的*copy_from_user*、*kmalloc*、*printk*。如果用户确实需要程序休眠，请首先通过调用*in_interrupt*来判断驱动程序是否在中断上下文中。

下面的示例驱动程序并没有连接到真正的硬件上，因此它的*write*函数只是简单地在内核调试日志中记录了需要写入的数据。请看下面的代码:

```
static int tiny_write(struct tty_struct *tty,
                const unsigned char *buffer, int count)
{
    struct tiny_serial *tiny = tty->driver_data;
    int i;
    int retval = -EINVAL;

    if (!tiny)
        return -ENODEV;

    down(&tiny->sem);
    if (!tiny->open_count)
        /* 端口未被打开 */
        goto exit;

     /* 将数据写入内核调试日志，来伪装将数据发送出硬件端口 */

    printk(KERN_DEBUG "%s - ", __FUNCTION__);
    for (i = 0; i < count; ++i)
        printk("%02x ", buffer[i]);
    printk("\n");

exit:
    up(&tiny->sem);
    return retval;
}
```

当tty子系统本身需要将一些数据传送到tty设备之外时，可以调用*write*函数。如果tty驱动程序并未在tty_struct中实现*put_char*函数，将会发生这类操作。此时，tty核心使用数据大小为1的参数回调*write*函数。这通常发生在tty核心将换行字符转换成一

个换行字符和一个新行字符时。这么做的最大问题是：在发生这种调用时，tty驱动程序的*write*函数一定不能返回0。这就意味着当调用者（tty核心）并未缓冲数据，而其后又一次执行时，驱动程序必须向设备写入一个字节的数据。由于*write*函数并不能判定它是否在*put_char*处被调用，即使只发送一个字节的数据，在实现*write*函数时，应在返回之前至少写入一个字节。现在许多USB到串口的tty驱动程序并不依照此规则行事，因此当连接到一些类型的终端时，它们就不能正常工作。

当tty核心想知道由tty驱动程序提供的可用写入缓冲区大小时，就会调用*write_room*函数。在清空写缓冲区，或者调用*write*函数向缓冲区添加数据时，该值是变化的。

```
static int tiny_write_room(struct tty_struct *tty)
{
    struct tiny_serial *tiny = tty->driver_data;
    int room = -EINVAL;

    if (!tiny)
        return -ENODEV;

    down(&tiny->sem);

    if (!tiny->open_count) {
         /* 端口没有被打开 */
        goto exit;
    }
    /* 计算设备中可用空间的大小 */
    room = 255;

exit:
    up(&tiny->sem);
    return room;
}
```

其他缓冲函数

为了获取正在工作的tty驱动程序，在tty_driver中的*chars_in_buffer*函数并不是必需的，但是推荐使用。当tty核心想知道在tty驱动程序的写缓冲区中还有多少个需要传输的字符时调用该函数。如果驱动程序在将字符发送给硬件设备前能够保存它们，则需要实现该函数，这样tty核心就能判断在驱动程序中的数据是否都被传输了。

在tty_driver中有三个回调函数用来刷新驱动程序保存的任何数据。它们并不需要一定被实现，但是如果tty驱动程序能在发送给硬件前缓冲数据，还是推荐实现它们。前两个回调函数是*flush_chars*和*wait_until_sent*。当tty核心使用*put_char*回调函数把大量字符发送给tty驱动程序时调用它们。如果tty驱动程序还没有开始发送数据，则内核要求tty驱动程序开始把数据发送给硬件，此时使用*flush_chars*回调函数。在将全部的数据发送给硬件前，该函数可以返回。*wait_until_sent*回调函数与此非常类似，但只有

当所有的数据都被发送给tty核心，或者超过了设定的超时值时，其才能返回。tty驱动程序在函数执行时可以休眠以完成操作。如果传递给*wait_until_sent*函数的超时值为0，则其只能等待操作完成后才可返回。

剩下的一个刷新回调函数是*flush_buffer*。当tty驱动程序要刷新在其写缓冲区中的所有数据时，tty核心调用该函数。此时保存在缓冲区中的所有数据都将被删除，而不能发送给设备。

怎么没有read函数？

只使用这些函数，*tiny_tty*驱动程序可以进行注册，一个设备节点可以被打开，数据可以被写入设备，设备节点可以被关闭，驱动程序可以注销并从内核中卸载。但是tty核心和*tty_driver*结构并未提供read函数；换句话说，并不存在一个回调函数从驱动程序获得数据并传递给tty核心。

当tty驱动程序接收到数据后，它将负责把从硬件获取的任何数据传递给tty核心，而不使用传统的read函数。tty核心将缓冲数据直到接到来自用户的请求。由于tty核心已提供了缓冲逻辑，因此没有必要为每个tty驱动程序实现它们自己的缓冲区逻辑。当用户要求驱动程序开始或者停止传输数据时，tty核心将通知tty驱动程序。但是如果内部的tty缓冲区已经写满，将不会有上述的通知事件。

在一个名为tty_flip_buffer的结构中，tty核心缓冲从tty驱动程序接收的数据。交替（flip）缓冲区是一个含有两个主要数据数组的结构。从tty接收到的数据保存在第一个数组中。当该数组存满后，等待这些数据的用户将被告之：数据已就绪可以读取了。当用户从这个数组中读数据时，任何新接收的数据将被保存到第二个数组中。当这个数组存满后，数据又一次地流向用户，而此时驱动程序开始填充第一个数组。因此被接收的数据从一个缓冲区交替保存到另外一个缓冲区，只是希望两个缓冲区不要同时溢出。为了保护数据不至丢失，tty驱动程序可以监控输入数组的大小，如果已经满了，则tty驱动程序及时刷新缓冲区，而不是等待下一次更新的机会。

如果使用了变量count，那么tty_flip_buffer结构的技术细节对tty驱动程序并非至关重要。这个变量保存了当前缓冲区中用于接收数据的剩余字节数。如果该变量取值TTY_FLIPBUF_SIZE，那么需要调用*tty_flip_buffer_push*函数将交替缓冲区中的数据发送给用户。该过程的例子代码如下：

```
for (i = 0; i < data_size; ++i) {
    if (tty->flip.count >= TTY_FLIPBUF_SIZE)
        tty_flip_buffer_push(tty);
    tty_insert_flip_char(tty, data[i], TTY_NORMAL);
}
tty_flip_buffer_push(tty);
```

调用*tty_insert_flip_char*将把tty驱动程序获得的、准备发给用户的字符添加到交替缓冲区中。该函数的第一个参数是保存数据的tty_struct结构，第二个参数是需要保存的数据，第三个参数是为此字符设置的标志位。如果接收的字符是常规字符，标志位应该被设置为TTY_NORMAL。如果是一个特殊类型的字符表示产生了接收数据错误，这些字符根据不同的错误可能是TTY_BREAK、TTY_FRAME、TTY_PARITY或者是TTY_OVERRUN。

为了把数据“推”向用户，需要调用*tty_flip_buffer_push*。当交替缓冲区将要溢出时，也调用这个函数，可以在例子程序中看到这点。因此无论何时把数据添加到交替缓冲区，或者交替缓冲区被占满，tty驱动程序一定要调用*tty_flip_buffer_push*。如果tty驱动程序能够以很快的速度接收数据，则需要设置tty->low_latency标志位，这使得*tty_flip_buffer_push*函数在被调用时，会立刻被执行。否则*tty_flip_buffer_push*会在未来的某个时刻才会将数据清除出缓冲区。

TTY 线路设置

当用户要改变线路设置，或者获得当前的线路设置，只需要调用多个termios用户空间库函数中的一个，也可直接对tty设备节点调用*ioctl*。tty核心将会把这两种接口转换为一系列的tty驱动程序的回调函数，或者是*ioctl*调用。

set_termios

大部分termios的用户空间函数将会被库转换成对驱动程序节点的*ioctl*调用。大量的不同tty *ioctl*调用会被tty核心转换成一个对tty驱动程序的*set_termios*函数调用。*set_termios*回调函数需要知道要改变的是哪一个线路设置，然后在tty设备中对其进行改动。tty驱动程序必须能够对在*termios*结构中所有不同的设置进行解码，并对任何需要的改变做出响应。因为所有的线路设置都被封装在termios结构中，因此这是个复杂的工作。

*set_termios*函数首先要做的是判断是否需要改变设置。可以使用下面的代码进行判断：

```
unsigned int cflag;

cflag = tty->termios->c_cflag;

/* 检查以保证确实有参数要被改变 */
if (old_termios) {
    if ((cflag =  = old_termios->c_cflag) &&
        (RELEVANT_IFLAG(tty->termios->c_iflag) =  =
         RELEVANT_IFLAG(old_termios->c_iflag))) {
        printk(KERN_DEBUG " - nothing to change...\n");
        return;
    }
}
```

宏RELEVANT_IFLAG的定义如下:

```
#define RELEVANT_IFLAG(iflag) ((iflag) & (IGNBRK|BRKINT|IGNPAR|PARMRK|INPCK))
```

该宏可以用来屏蔽cflag变量中重要的位。通过与原来的值进行比较，来发现是否产生变化。如果没有变化，则不需要改变任何设置，接着返回。请注意，在访问old_termios变量前首先应检查它是否是个合法的指针。由于有时该变量被设置为NULL，所以检查是必需的。如果对一个NULL指针进行访问，会产生内核panic。

为了获得所需要的字节大小，可以使用CSIZE掩码把正确的位从cflag变量中分离出去。如果不确定字节大小，习惯上将被设置为默认的8数据位。请参看下面的执行代码:

```
/* 获得字节大小 */
switch (cflag & CSIZE) {
    case CS5:
        printk(KERN_DEBUG " - data bits = 5\n");
        break;
    case CS6:
        printk(KERN_DEBUG " - data bits = 6\n");
        break;
    case CS7:
        printk(KERN_DEBUG " - data bits = 7\n");
        break;
    default:
    case CS8:
        printk(KERN_DEBUG " - data bits = 8\n");
        break;
}
```

为了确定所需要的奇偶校验值，对cflag变量使用PARENB掩码，可以知道是否设置了奇偶校验。如果设置了奇偶校验，使用PARODD掩码能够判断使用的是奇校验还是偶校验。下面是一个例子:

```
/* 判断奇偶 */
if (cflag & PARENB)
    if (cflag & PARODD)
        printk(KERN_DEBUG " - parity = odd\n");
    else
        printk(KERN_DEBUG " - parity = even\n");
else
    printk(KERN_DEBUG " - parity = none\n");
```

对cflag变量使用CSTOPB位能够确定出是否使用停止位。下面是例子代码:

```
/* 确定需要的停止位 */
if (cflag & CSTOPB)
    printk(KERN_DEBUG " - stop bits = 2\n");
else
    printk(KERN_DEBUG " - stop bits = 1\n");
```

有两种基本类型的流控制：软件控制和硬件控制。对cflag使用CRTSCTS掩码能确定用户是否使用的是硬件流控制。示例代码如下：

```
/* 确定硬件流控制设置 */
if (cflag & CRTSCTS)
    printk(KERN_DEBUG " - RTS/CTS is enabled\n");
else
    printk(KERN_DEBUG " - RTS/CTS is disabled\n");
```

确定使用软件流控制的不同模式，以及使用不同的开始和停止字符稍微有点麻烦：

```
/* 确定软件流控制 */
/* 如果实现了XON/XOFF，设置设备中的开始及结束字符 */
if (I_IXOFF(tty) || I_IXON(tty)) {
    unsigned char stop_char  = STOP_CHAR(tty);
    unsigned char start_char = START_CHAR(tty);

    /* 如果实现了INBOUND XON/XOFF */

    if (I_IXOFF(tty))
        printk(KERN_DEBUG " - INBOUND XON/XOFF is enabled, "
            "XON = %2x, XOFF = %2x", start_char, stop_char);
    else
        printk(KERN_DEBUG" - INBOUND XON/XOFF is disabled");

    /* 如果实现了OUTBOUND XON/XOFF */
    if (I_IXON(tty))
        printk(KERN_DEBUG" - OUTBOUND XON/XOFF is enabled, "
            "XON = %2x, XOFF = %2x", start_char, stop_char);
    else
        printk(KERN_DEBUG" - OUTBOUND XON/XOFF is disabled");
}
```

最后要确定使用的波特率。tty 核心提供了函数 *tty_get_baud_rate* 以达到此目的。该函数返回一个整型值表示对特殊的 tty 设备所使用的波特率。

```
/* 获得需要的波特率 */
printk(KERN_DEBUG " - baud rate = %d", tty_get_baud_rate(tty));
```

现在 tty 驱动程序可以确定所有的线路设置，可以使用这些值正确启动硬件。

tiocmget 和 tiocmset

在2.4及更早的内核中，使用了大量的tty *ioctl*调用来获得及设置不同的控制线路参数。这通过常量TIOCMGET、TIOCMBIS、TIOCMBIC和TIOCMSET来完成。TIOCMGET用来获得内核的线路设置值，在2.6版本的内核中，该*ioctl*调用被tty驱动程序中的*tiocmget*回调函数所代替。剩下的三个*ioctl*现在被简化成tty驱动程序中的一个*tiocmset*回调函数了。

当内核要了解特定tty设备控制线路的当前物理值时，tty核心会调用tty驱动程序中的*tiocmget*函数。这常用于获得串口DTR和RTS控制线的值。如果由于硬件不支持，tty驱动程序无法直接读取串口的MSR或者MCR寄存器，将在本地保存这两个值的副本。许多USB到串口驱动程序必须实现这类监控操作。下面是当需要在本地保存这些值的副本时，实现该函数的一个例子：

```
static int tiny_tiocmget(struct tty_struct *tty, struct file *file)
{
    struct tiny_serial *tiny = tty->driver_data;

    unsigned int result = 0;
    unsigned int msr = tiny->msr;
    unsigned int mcr = tiny->mcr;

    result = ((mcr & MCR_DTR)  ? TIOCM_DTR  : 0) |  /* 设置了DTR */
             ((mcr & MCR_RTS)  ? TIOCM_RTS  : 0) |  /* 设置了RTS */
             ((mcr & MCR_LOOP) ? TIOCM_LOOP : 0) |  /* 设置了LOOP */
             ((msr & MSR_CTS)  ? TIOCM_CTS  : 0) |  /* 设置了CTS */
             ((msr & MSR_CD)   ? TIOCM_CAR  : 0) |  /* 设置了Carrier detect */
             ((msr & MSR_RI)   ? TIOCM_RI   : 0) |  /* 设置了Ring Indicator */
             ((msr & MSR_DSR)  ? TIOCM_DSR  : 0);   /* 设置了DSR */
    return result;
}
```

当tty核心要为一个特定的tty设备设置控制线路值时，它将调用tty驱动程序中的*tiocmset*函数。tty核心通过传递两个变量set和clear，告诉tty驱动程序设置成什么值以及清除哪一位。这些变量包含了将要改变的线路设置的一个掩码。一个*ioctl*调用从不要求tty驱动程序同时设置和清除某一位，因此哪个操作先进行是无所谓的。下面是该函数tty驱动程序实现的例子代码：

```
static int tiny_tiocmset(struct tty_struct *tty, struct file *file,
                         unsigned int set, unsigned int clear)
{
    struct tiny_serial *tiny = tty->driver_data;
    unsigned int mcr = tiny->mcr;

    if (set & TIOCM_RTS)
        mcr |= MCR_RTS;
    if (set & TIOCM_DTR)
        mcr |= MCR_RTS;

    if (clear & TIOCM_RTS)
        mcr &= ~MCR_RTS;
    if (clear & TIOCM_DTR)
        mcr &= ~MCR_RTS;

    /* 设置设备中新的MCT值 */
    tiny->mcr = mcr;
    return 0;
}
```

ioctls

当 *ioctl(2)*为一个设备节点被调用时，tty 核心将调用 tty_driver 结构中的 *ioctl* 回调函数。如果 tty 驱动程序不知道如何处理传递给它的 *ioctl* 值，它可返回 -ENOIOCTLCMD，从而让 tty 核心执行一个通用的操作。

在 2.6 版本的内核中定义了 70 个可以发送给 tty 驱动程序的不同的 tty *ioctl*。大多数 tty 驱动程序并不能处理所有这些 *ioctl*，而只能处理一小部分常用的调用。下面是一个常用 tty *ioctl* 的列表，我们说明了它们的含义和如何实现它们：

TIOCSERGETLSR

获得这个 tty 设备线路状态寄存器（LSR）的值。

TIOCGSERIAL

获得串行线路信息。使用该调用，可以从 tty 设备那里一次获得许多串行线路的信息。一些程序（比如 *setserial* 和 *dip*）调用这个函数以确定正确设置了波特率，并且获得 tty 驱动程序所控制的设备类型一般信息。调用者传递进一个指向 serial_struct 结构的指针，tty 驱动程序要为其填充正确的值。下面是这个过程的实现代码：

```
static int tiny_ioctl(struct tty_struct *tty, struct file *file,
                      unsigned int cmd, unsigned long arg)
{
    struct tiny_serial *tiny = tty->driver_data;
    if (cmd == TIOCGSERIAL) {
        struct serial_struct tmp;
        if (!arg)
            return -EFAULT;
        memset(&tmp, 0, sizeof(tmp));
        tmp.type        = tiny->serial.type;
        tmp.line        = tiny->serial.line;
        tmp.port        = tiny->serial.port;
        tmp.irq         = tiny->serial.irq;
        tmp.flags       = ASYNC_SKIP_TEST | ASYNC_AUTO_IRQ;
        tmp.xmit_fifo_size  = tiny->serial.xmit_fifo_size;
        tmp.baud_base       = tiny->serial.baud_base;
        tmp.close_delay     = 5*HZ;
        tmp.closing_wait    = 30*HZ;
        tmp.custom_divisor  = tiny->serial.custom_divisor;
        tmp.hub6        = tiny->serial.hub6;
        tmp.io_type     = tiny->serial.io_type;
        if (copy_to_user((void __user *)arg, &tmp, sizeof(tmp)))
            return -EFAULT;
        return 0;
    }
    return -ENOIOCTLCMD;
}
```

TIOCSSERIAL

设置串行线路信息。它与TIOCGSERIAL相反，允许用户同时设置tty设备的串行线路状态。一个指向serial_struct结构的指针传递给该调用，里面包含了tty设备需要设置的信息。如果tty驱动程序没有实现这一调用，大多数程序依然能工作正常。

TIOCMIWAIT

等待MSR的变化。用户在非常规环境下调用该功能，该功能将在内核中休眠，直到有事件改变了tty设备的MSR寄存器为止。arg参数包含了用户等待的事件类型。它常被用来处于等待状态直到状态连接有所变化，然后报告数据已经就绪并可发送给设备了。

实现这个*ioctl*时一定要谨慎，不要使用*interruptible_sleep_on*调用，因为该调用并不安全（使用它将会产生大量竞态问题）。相反，使用*wait_queue*可以避免这一问题。下面是实现这个*ioctl*的例子：

```
static int tiny_ioctl(struct tty_struct *tty, struct file *file,
                      unsigned int cmd, unsigned long arg)
{
    struct tiny_serial *tiny = tty->driver_data;
    if (cmd == TIOCMIWAIT) {
        DECLARE_WAITQUEUE(wait, current);
        struct async_icount cnow;
        struct async_icount cprev;
        cprev = tiny->icount;
        while (1) {
            add_wait_queue(&tiny->wait, &wait);
            set_current_state(TASK_INTERRUPTIBLE);
            schedule();
            remove_wait_queue(&tiny->wait, &wait);
            /* 检查是否有信号将代码唤醒 */

            if (signal_pending(current))
                return -ERESTARTSYS;
            cnow = tiny->icount;
            if (cnow.rng == cprev.rng && cnow.dsr == cprev.dsr &&
                cnow.dcd == cprev.dcd && cnow.cts == cprev.cts)
                return -EIO; /* 没有变化 => 错误 */
            if (((arg & TIOCM_RNG) && (cnow.rng != cprev.rng)) ||
                ((arg & TIOCM_DSR) && (cnow.dsr != cprev.dsr)) ||
                ((arg & TIOCM_CD)  && (cnow.dcd != cprev.dcd)) ||
                ((arg & TIOCM_CTS) && (cnow.cts != cprev.cts)) ) {
                return 0;
            }
            cprev = cnow;
        }
    }
    return -ENOIOCTLCMD;
}
```

如果侦测出MSR寄存器发生变化，在tty驱动程序代码的某处，要使用下面的代码以使驱动工作正常：

```
wake_up_interruptible(&tp->wait);
```

TIOCGICOUNT

获得中断计数。当用户想知道发生了多少次串行线路中断时，使用该调用。如果驱动程序有中断处理的话，则要定义一个内部的计数器数据结构来记录这些统计值，并在内核调用这个函数的时候，增加正确的值。

这个*ioctl*调用向内核传递一个指向serial_icounter_struct结构的指针，该结构应该由tty驱动程序填写。该调用经常与前面的TIOCMIWAIT *ioctl*调用联合使用。如果tty驱动程序运行时监控并保存了所有这些中断，实现该调用的代码非常简单：

```
static int tiny_ioctl(struct tty_struct *tty, struct file *file,
                      unsigned int cmd, unsigned long arg)
{
    struct tiny_serial *tiny = tty->driver_data;
    if (cmd == TIOCGICOUNT) {
        struct async_icount cnow = tiny->icount;
        struct serial_icounter_struct icount;
        icount.cts   = cnow.cts;
        icount.dsr   = cnow.dsr;
        icount.rng   = cnow.rng;
        icount.dcd   = cnow.dcd;
        icount.rx    = cnow.rx;
        icount.tx    = cnow.tx;
        icount.frame     = cnow.frame;
        icount.overrun   = cnow.overrun;
        icount.parity    = cnow.parity;
        icount.brk   = cnow.brk;
        icount.buf_overrun = cnow.buf_overrun;
        if (copy_to_user((void __user *)arg, &icount, sizeof(icount)))
            return -EFAULT;
        return 0;
    }
    return -ENOIOCTLCMD;
}
```

proc 和 sysfs 对 TTY 设备的处理

tty 核心为任何 tty 驱动程序都提供了非常简单的办法，用来维护在 */proc/tty/driver* 目录中的一个文件。如果驱动程序定义了 *read_proc* 或者 *write_proc* 函数，将创建该文件。接着任何对该文件的读写将被发送给驱动程序。这些函数的格式与标准的 */proc* 文件处理函数相同。

这里有个例子，它简单地实现了*read_proc*回调函数，只是用来打印出当前注册的端口：

```
static int tiny_read_proc(char *page, char **start, off_t off, int count,
                          int *eof, void *data)
{
    struct tiny_serial *tiny;
    off_t begin = 0;
    int length = 0;
    int i;

    length += sprintf(page, "tinyserinfo:1.0 driver:%s\n", DRIVER_VERSION);
    for (i = 0; i < TINY_TTY_MINORS && length < PAGE_SIZE; ++i) {
        tiny = tiny_table[i];
        if (tiny = = NULL)
            continue;

        length += sprintf(page+length, "%d\n", i);
        if ((length + begin) > (off + count))
            goto done;
        if ((length + begin) < off) {
            begin += length;
            length = 0;
        }
    }
    *eof = 1;
done:
    if (off >= (length + begin))
        return 0;
    *start = page + (off-begin);
    return (count < begin+length-off) ? count : begin + length-off;
}
```

当根据tty_driver结构中的TTY_DRIVER_NO_DEVFS标志注册tty驱动程序时，或者创建一个单独的tty设备时，tty核心处理所有的sysfs目录和设备的创建。单独的目录总是包含*dev*文件，这可以让用户空间的工具来判定分配给该设备的主设备号和次设备号。如果在调用*tty_register_device*函数时使用了合法的*device*结构，它还将包含一个设备和驱动程序的符号连接。除了这三个文件外，无法为单独的tty驱动程序在该目录下创建新的sysfs文件。在未来的内核发行版中，这点可能会有所改变。

tty_driver结构详解

tty_driver结构用来向tty核心注册一个tty驱动程序。下面是一个该结构所有成员以及它们如何被tty核心使用的列表：

struct module *owner;

该驱动程序模块的所有者。

`int magic;`

该结构的“magic(幻数)”值。通常被设置为TTY_DRIVER_MAGIC。在*alloc_tty_driver*函数中被初始化。

`const char *driver_name;`

在*/proc/tty*和sysfs中使用，表示驱动程序的名字。

`const char *name;`

驱动程序节点的名字。

`int name_base;`

为创建设备名字而使用的开始编号。当内核创建一个分配给tty驱动程序的、表示特定tty设备的名称时使用。

`short major;`

驱动程序的主设备号。

`short minor_start;`

驱动程序使用的最小次设备号。通常设置成与name_base相同的值。该值一般设为0。

`short num;`

可以分配给驱动程序次设备号的个数。如果驱动程序使用了全部范围的主设备号，该值要被设置为255。该变量在*alloc_tty_driver*函数中被初始化。

`short type;`

`short subtype;`

描述向tty核心注册的是何种tty驱动程序。subtype取决于type。type的值可以为:

TTY_DRIVER_TYPE_SYSTEM

在tty子系统内部使用，表明其正在处理一个内部的tty驱动程序。subtype要被设置为SYSTEM_TYPE_TTY、SYSTEM_TYPE_CONSOLE、SYSTEM_TYPE_SYSCONS或者是SYSTEM_TYPE_SYSPTMX。这种类型不能被“常规”的tty驱动程序使用。

TTY_DRIVER_TYPE_CONSOLE

只被控制台驱动程序使用。

TTY_DRIVER_TYPE_SERIAL

可以被任何串行类驱动程序使用。subtype可以设置为SERIAL_TYPE_NORMAL或者SERIAL_TYPE_CALLOUT，这取决于驱动程序的类型。这是type最常使用的设置之一。

TTY_DRIVER_TYPE_PTY
被伪终端接口（pty）所使用。subtype需要被设置为PTY_TYPE_MASTER或者PTY_TYPE_SLAVE。

struct termios init_termios;
当被创建时，含有初始值的termios结构。

int flags;
驱动程序标志位，如本章前面所述。

struct proc_dir_entry *proc_entry;
该驱动程序的/proc入口结构体。如果驱动程序实现*write_proc*或者*read_proc*，它将由tty核心创建。该值不能由tty驱动程序本身设置。

struct tty_driver *other;
指向tty从属设备驱动程序的指针。它只能被pty驱动程序使用，而不能被任何其他的tty驱动程序使用。

void *driver_state;
tty驱动程序内部的状态。只能被pty驱动程序使用。

struct tty_driver *next;
struct tty_driver *prev;
链接变量。这些变量被tty核心使用，把所有不同的tty驱动程序链接起来，并且不能被任何tty驱动程序访问。

tty_operations结构详解

tty_operations结构中包含所有的回调函数，它们被tty驱动程序设置，并被tty核心调用。目前，该结构所包含的所有函数指针也包含在tty_driver结构中，但这很快会被替代，因为只能有该结构的一个实例存在。

int (*open)(struct tty_struct * tty, struct file * filp);
*open*函数。

void (*close)(struct tty_struct * tty, struct file * filp);
*close*函数。

int (*write)(struct tty_struct * tty, const unsigned char *buf, int count);
*write*函数。

```
void (*put_char)(struct tty_struct *tty, unsigned char ch);
```

单字符写入函数。当要把一个字符写入设备时，该函数被 tty 核心调用。如果一个 tty 驱动程序没有定义这个函数，当 tty 核心要发送一个字符时，用 *write* 函数作为替代。

```
void (*flush_chars)(struct tty_struct *tty);
void (*wait_until_sent)(struct tty_struct *tty, int timeout);
```

该函数用来向硬件发送数据。

```
int (*write_room)(struct tty_struct *tty);
```

该函数用来检测缓冲区的剩余空间。

```
int (*chars_in_buffer)(struct tty_struct *tty);
```

该函数用来检测包含数据的缓冲区数量。

```
int (*ioctl)(struct tty_struct *tty, struct file * file, unsigned int cmd,
            unsigned long arg);
```

ioctl 函数。当对设备节点调用 *ioctl(2)*时，该函数被 tty 核心调用。

```
void (*set_termios)(struct tty_struct *tty, struct termios * old);
```

set_termios 函数。当设备的 termios 设置发生改变时，该函数被 tty 核心调用。

```
void (*throttle)(struct tty_struct * tty);
void (*unthrottle)(struct tty_struct * tty);
void (*stop)(struct tty_struct *tty);
void (*start)(struct tty_struct *tty);
```

数据控制函数。这些函数用来控制并防止 tty 核心的输入缓冲区溢出。当 tty 核心的输入缓冲区满的时候，调用 *throttle* 函数。tty 驱动程序将试图通知设备，不要再发送更多的字符了。当 tty 核心的输入缓冲区被清空时，调用 *unthrottle* 函数，使其能接受更多的数据。tty 驱动程序将通知设备，它可以接收字符了。*stop* 和 *start* 函数和 *throttle* 与 *unthrottle* 函数相似，但是它们表示：tty 驱动程序将停止向设备发送数据，并在未来恢复数据的传送。

```
void (*hangup)(struct tty_struct *tty);
```

挂起函数。当 tty 驱动程序挂起 tty 设备时，调用该函数。此时对任何特定硬件的操纵应当被挂起。

```
void (*break_ctl)(struct tty_struct *tty, int state);
```

中断连接控制函数。当 tty 驱动程序要打开或者关闭 RS-232 端口的 BREAK 线路状态时被调用。如果状态被设置为 -1，则 BREAK 权限应该被打开。如果状态被设置为 0，BREAK 权限应该被关闭。如果 tty 驱动程序实现了该函数，tty 核心将处

理 TCSBRK、TCSBRKP、TIOCSBRK 和 TIOCCBRK *ioctl*。否则这些 *ioctl* 会被发送给驱动程序中的 *ioctl* 函数。

```
void (*flush_buffer)(struct tty_struct *tty);
```

刷新缓冲区，并丢失里面的数据。

```
void (*set_ldisc)(struct tty_struct *tty);
```

设置线路规程的函数。当 tty 核心改变了 tty 驱动程序的线路规程时调用它。该函数通常不能被驱动程序使用，也不应该由驱动程序来定义。

```
void (*send_xchar)(struct tty_struct *tty, char ch);
```

发送 X 类型字符函数。该函数用来向 tty 设备发送高优先级的 XON 或者 XOFF 字符。要发送的字符放在 ch 变量中。

```
int (*read_proc)(char *page, char **start, off_t off, int count, int *eof,
                 void *data);
int (*write_proc)(struct file *file, const char *buffer, unsigned long count,
                  void *data);
```

/proc 的 *read* 和 *write* 函数。

```
int (*tiocmget)(struct tty_struct *tty, struct file *file);
```

获得特定 tty 设备当前的线路设置。如果能成功地从 tty 设备获得设置，该值将被返回给调用者。

```
int (*tiocmset)(struct tty_struct *tty, struct file *file, unsigned int set,
                unsigned int clear);
```

为特定的 tty 设备设置当前线路。set 和 clear 包含了将要被设置或者清除的线路设置。

tty_struct 结构详解

tty 核心使用 tty_struct 保存当前特定 tty 端口的状态。除了少数例外，该结构中几乎所有的成员都只能被 tty 核心使用。tty 驱动程序可以使用的成员描述如下：

```
unsigned long flags;
```

当前 tty 设备的状态。它是个位操作变量，可以通过下面的宏来访问该值：

TTY_THROTTLED

当驱动程序调用 *throttle* 函数时设置该值。只有 tty 核心，而不是 tty 驱动程序，能够设置该值。

TTY_IO_ERROR

当驱动程序禁止向其读写数据时设置该值。如果一个用户程序要这么做，那么它将从内核接收到一个 -EIO 错误。通常在关闭设备时设置该值。

TTY_OTHER_CLOSED

通知端口已被关闭，只能由 pty 驱动程序使用。

TTY_EXCLUSIVE

由 tty 核心设置，表示端口处于独占模式，一次只能有一个用户访问它。

TTY_DEBUG

在内核中不使用。

TTY_DO_WRITE_WAKEUP

如果设置该值，则允许调用线路规程的*write_wakeup*函数。通常 tty 驱动程序会同时调用 *wake_up_interruptible* 函数。

TTY_PUSH

仅在内部被默认的 tty 线路规程使用。

TTY_CLOSING

tty 核心使用它监控端口此时是否正处在关闭过程中。

TTY_DONT_FLIP

被默认的 tty 线路规程使用。当其被设置时，用来通知 tty 核心，不能改变交替缓冲区。

TTY_HW_COOK_OUT

如果 tty 驱动程序设置它，驱动程序将通知线路规程：将要“修改”发送给它的数据。如果没有被设置，线路规程将成块地拷贝 tty 驱动程序的输出；否则它将评估修改线路设置的每一个单独发送的字节。该标志通常不能由 tty 驱动程序设置。

TTY_HW_COOK_IN

几乎与设置驱动程序flags变量为TTY_DRIVER_REAL_RAW的情况相同。该标志通常不能由 tty 驱动程序设置。

TTY_PTY_LOCK

pty 驱动程序使用它来锁住和解锁端口。

TTY_NO_WRITE_SPLIT

如果设置该位，tty 核心不能将数据分割成常规大小发送给 tty 驱动程序。不使用该值，可以防止向 tty 端口发送大量数据的拒绝服务攻击。

`struct tty_flip_buffer flip;`

tty 设备的交替缓冲区。

`struct tty_ldisc ldisc;`

tty 设备的线路规程。

`wait_queue_head_t write_wait;`

用于 tty 写函数的 *wait_queue*。当一个 tty 驱动程序可以接收数据时，应当唤醒该队列。

`struct termios *termios;`

指向设置 tty 设备的 termios 结构指针。

`unsigned char stopped:1;`

表示 tty 设备是否已经停止。tty 驱动程序可以设置该值。

`unsigned char hw_stopped:1;`

表示 tty 设备硬件是否已经停止。tty 驱动程序可以设置该值。

`unsigned char low_latency:1;`

表示 tty 设备是否是个慢速设备，是否能接收高速传输的数据。tty 驱动程序可以设置该值。

`unsigned char closing:1;`

表示 tty 设备是否正在关闭端口。tty 驱动程序可以设置该值。

`struct tty_driver driver;`

控制 tty 设备的当前 `tty_driver` 结构。

`void *driver_data;`

`tty_driver` 用来把数据保存在 tty 驱动程序中的指针。该变量不能由 tty 核心来改变。

快速参考

本节提供了一些本章所讲述概念的参考介绍。它还介绍了 tty 驱动程序所需要的各个头文件的作用。当然 `tty_driver` 和 `tty_device` 结构中的每一个成员，这里就不再重复了。

`#include <linux/tty_driver.h>`

包含 `tty_driver` 结构定义，以及在该结构中一些不同标志位的声明。

#include <linux/tty.h>

该头文件包含了tty_struct结构的定义以及许多不同的宏定义，使得对termios结构各成员值的访问更简单。它还包含了 tty 驱动程序核心的函数声明。

#include <linux/tty_flip.h>

包含了一些 tty 交替缓冲区 inline 函数的头文件，这些 inline 函数能简化对交替缓冲区结构的操作。

#include <asm/termios.h>

特定硬件平台创建内核时，使用的是包含 termio 结构定义的头文件。

struct tty_driver *alloc_tty_driver(int lines);

创建 tty_driver 结构的函数，该结构以后将被传递给 *tty_register_driver* 和 *tty_unregister_driver* 函数。

void put_tty_driver(struct tty_driver *driver);

如果使用tty_driver结构向tty核心的注册没有成功，该函数负责清空 tty_driver 结构。

void tty_set_operations(struct tty_driver *driver, struct tty_operations *op);

负责初始化结构 tty_driver 中的回调函数的函数。在调用 *tty_register_driver* 前必须调用它。

int tty_register_driver(struct tty_driver *driver);
int tty_unregister_driver(struct tty_driver *driver);

从 tty 核心注册和注销一个 tty 驱动程序的函数。

void tty_register_device(struct tty_driver *driver, unsigned minor, struct device *device);
void tty_unregister_device(struct tty_driver *driver, unsigned minor);

向 tty 核心注册和注销一个 tty 设备的函数。

void tty_insert_flip_char(struct tty_struct *tty, unsigned char ch, char flag);

将字符插入到 tty 设备的交替缓冲区使用户能够读到的函数。

TTY_NORMAL
TTY_BREAK
TTY_FRAME
TTY_PARITY
TTY_OVERRUN

在 *tty_insert_flip_char* 函数中使用的不同的标志位。

`int tty_get_baud_rate(struct tty_struct *tty);`

获得指定tty设备当前波特率的函数。

`void tty_flip_buffer_push(struct tty_struct *tty);`

把数据放入当前交替缓冲区并传递给用户的函数。

`tty_std_termios`

用常见的默认线路设置初始化termios结构的变量。

参考书目

本书的大部分内容来自内核源代码，而内核源代码是有关 Linux 内核的最好文档。

我们可以从遍布全球的上百个FTP站点上获得内核源代码，因此，我们不打算在这里列出这些站点。

通过观察补丁，我们可很好地检查版本依赖性，这些补丁通常位于下载得到整个源代码的同一站点。我们还可以通过 repatch 程序来检查单个文件在不同内核补丁中是如何被修改的，该工具的源代码可在 O'Reilly FTP 站点上获得。

书籍

书店的书架上充斥着大量技术书籍，但是只有少量书籍直接和Linux内核编程相关。下面是出现在作者自己书架上的书籍清单：

Linux 内核相关书籍

作者：Daniel P. Bovet 和 Marco Cesate。

Understanding the Linux Kernel, Second Edition

出版商：Sebastopol, CA: O'Reilly Media, Inc. 2003.

该书详细讲述了Linux内核的设计和实现，它主要讲述内核使用的算法，而不是用来说明内核 API 的。虽然该书阐述的是 2.4 内核，但仍包含了大量有用信息。

作者：Mel Gorman。

Understanding the Linux Virtual Memory Manager

出版商：Upper Saddle River, NJ: Prentice Hall PTR, 2004.

希望更多了解 Linux 虚拟内存子系统的开发人员应该阅读该书。这本书主要讲述 2.4 内核，但同时包含 2.6 内核的相关信息。

作者：Robert Love。
Linux Kernel Development
出版商：Indianapolis: Sams Publishing, 2004.
该书描述了Linux内核编程的许多方面，每个Linux黑客都应该拥有这本书。

作者：Karim Yaghmour
Building Embedded Systems
出版商：Sebastopol, CA: O'Reilly & Associates, Inc. 2003.
对那些针对嵌入式系统编写Linux代码的人来讲，该书会非常有帮助。

Unix设计和内部实现相关的书籍

作者：Maurice Back
The Design of the Unix Operating System
出版商：Upper Saddle River, NJ: Prentice Hall, 1987.
尽管该书有点老了，但其中涵盖了大量Unix实现相关的问题。这本书也是Linus编写第一个Linux版本时的灵感来源。

作者：Richard Stevens
Advanced Programming in the UNIX Environment
出版商：Boston: Addison-Wesley, 1992.
该书非常详细地讲述了Unix系统调用，在实现设备方法时，这本书会是好的帮手。

作者：Richard Stevens
Unix Network Programming
出版商：Upper Saddle River, NJ: Prentice Hall PTR, 1990.
也许是讲述Unix网络编程API的最权威书籍。

Web站点

在Linux内核开发的快速变迁中最新的信息总能在线获得。下面是作者认为与本书相关的最好站点：

http://www.kernel.org
ftp://ftp.kernel.org
本站点是Linux内核开发的主站点，其中包含了最新的内核发行版本以及相关信息。注意该FTP站点的镜像遍布全球，因此，应该选择最近的镜像站点下载Linux源代码。

http://www.bkbits.net

该站点包含了大量优秀内核开发人员所使用的源代码仓库。特别是，称为“linus”的项目中包含了由Linus Torvalds维护的内核主线代码。如果读者对最近添加到内核中的补丁感兴趣，则应该访问这个站点。

http://www.tldp.org

“Linux Documentation Project（Linux文档项目）”站点拥有大量称作“HOWTO”的文档，其中一些是技术性的，并涉及到一些内核主题。

http://www.linux.it/kerneldocs

其中包含有许多Aleesandro Rubini所著的有关内核的杂志文章。某些文章在多年以前编写，但仍然有用；某些文章以意大利文编写，但通常也有英文翻译稿。

http://lwn.net

该新闻站点是自助式的，除了该站点提供的其他信息之外，最重要的是，它提供了定期的内核开发相关报道，以及API的修改信息。

http://www.kerneltraffic.org

“Kernel Traffic”是一个大众性的站点，它提供了每周Linux内核开发邮件列表中的讨论总结。

http://www.kerneltrap.org/

该站点报道了一些Linux和BSD内核社区的有趣开发活动。

http://www.kernelnewbies.org

该站点面向新的内核开发人员。其中包含有针对初学者的内容和FAQ，而且还有一个IRC频道，可获得即时的帮助。

http://janitor.kernelnewbies.org/

“Linux Kernel Janitor（Linux内核看门人）”项目，新的内核程序员可通过该站点了解到如何加入内核开发。该站点介绍了内核中大量需要完成的小的、通常比较简单的任务。还有一个邮件列表帮助新开发人员将这些改变加入到主流内核树中。对那些希望加入Linux内核开发，但又不知道从哪里入手的人来讲，这个站点再适合不过了。